LogiCORE™ IP Ethernet AVB Endpoint v2.4

User Guide

UG492 July 23, 2010

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
9/18/08	v1.1	Initial Xilinx release; ISE® 10.1, Update 3.
4/24/09	v1.2	Updated to version 1.2 of the core; Xilinx tools 11.1.
6/24/09	v2.1	Updated to version 2.1 of the core; Xilinx tools 11.2.
9/16/09	v2.2	Updated to version 2.2 of the core; Xilinx tools 11.3.
4/19/10	v2.3	Updated to version 2.3 of the core; Xilinx tools 12.1.
7/23/10	v2.4	Updated to version 2.4 of the core; Xilinx tools 12.2. Added four chapters from the Getting Started Guide to this User Guide: • Licensing the Core • Quick Start Example Design • Detailed Example Design (Standard Format) • Detailed Example Design (EDK format) The Getting Started Guide is being discontinued in this release.

Table of Contents

Chapter 4: Generating the Core

Chapter 5: Core Architecture

Chapter 6: Ethernet AVB Endpoint Transmission

Chapter 7: Ethernet AVB Endpoint Reception

Chapter 8: Real Time Clock and Time Stamping

Chapter 9: Precise Timing Protocol Packet Buffers

Chapter 10: Configuration and Status

Chapter 11: Constraining the Core

Chapter 12: System Integration

Chapter 13: Software Drivers

Chapter 14: Quick Start Example Design

Chapter 15: Detailed Example Design (Standard Format)

Chapter 16: Detailed Example Design (EDK format)

Appendix A: RTC Time Stamp Accuracy

Schedule of Figures

Chapter 1: Introduction

Chapter 2: Licensing the Core

Chapter 3: Overview of Ethernet Audio Video Bridging

Chapter 4: Generating the Core

Chapter 5: Core Architecture

Chapter 6: Ethernet AVB Endpoint Transmission

Chapter 7: Ethernet AVB Endpoint Reception

Chapter 8: Real Time Clock and Time Stamping

Chapter 9: Precise Timing Protocol Packet Buffers

Chapter 10: Configuration and Status

Chapter 11: Constraining the Core

Chapter 12: System Integration

Chapter 13: Software Drivers

Chapter 14: Quick Start Example Design

Chapter 15: Detailed Example Design (Standard Format)

Chapter 16: Detailed Example Design (EDK format)

Appendix A: RTC Time Stamp Accuracy

Schedule of Tables

Chapter 1: Introduction

Chapter 2: Licensing the Core

Chapter 3: Overview of Ethernet Audio Video Bridging

Chapter 4: Generating the Core

Chapter 5: Core Architecture

Chapter 6: Ethernet AVB Endpoint Transmission

Chapter 7: Ethernet AVB Endpoint Reception

Chapter 8: Real Time Clock and Time Stamping

Chapter 9: Precise Timing Protocol Packet Buffers

Chapter 10: Configuration and Status

Chapter 11: Constraining the Core

Chapter 12: System Integration

Chapter 13: Software Drivers

Chapter 14: Quick Start Example Design

Chapter 15: Detailed Example Design (Standard Format)

Chapter 16: Detailed Example Design (EDK format)

Appendix A: RTC Time Stamp Accuracy

About This Guide

The *LogiCORE™ IP Ethernet AVB User Guide* provides information about the Ethernet Audio Video Bridging (AVB) Endpoint core, including how to customize, generate, and implement the core in supported Xilinx FPGA families.

Guide Contents

This guide contains the following chapters:

- Preface, "About this Guide" introduces the organization and purpose of this guide and the conventions used in this document.
- Chapter 1, "Introduction" introduces the core and provides related information including additional core resources, technical support, and how to submit feedback to Xilinx.
- Chapter 2, "Licensing the Core" describes the available license options for the core and how to obtain them.
- Chapter 3, "Overview of Ethernet Audio Video Bridging" provides an overview of Ethernet Audio Video Bridging, including relevant specifications and a typical implementation.
- Chapter 4, "Generating the Core" provides information about generating and customizing the core using the CORE Generator™ software.
- Chapter 5, "Core Architecture" describes the major functional blocks of the Ethernet AVB Endpoint core.
- Chapter 6, "Ethernet AVB Endpoint Transmission" describes data transmission over an AVB network.
- Chapter 7, "Ethernet AVB Endpoint Reception" describes data reception over an AVB network.
- Chapter 8, "Real Time Clock and Time Stamping" describes two components that are partially responsible for the AVB timing synchronization protocol.
- Chapter 9, "Precise Timing Protocol Packet Buffers" describes two components that are partially responsible for the transmission and reception of Ethernet Precise Timing Protocol frames; these frames contain the AVB timing synchronization data.
- Chapter 10, "Configuration and Status" defines general guidelines for configuring and monitoring the Ethernet AVB Endpoint core, including an introduction to the PLB configuration bus and a description of the core management registers.
- Chapter 11, "Constraining the Core" defines the Ethernet AVB core constraints.
- Chapter 12, "System Integration" describes the integration of the Ethernet AVB Endpoint core into a system, including connection of the core to the Xilinx Tri-Mode Ethernet MAC and Ethernet Statistic cores.

- Chapter 13, "Software Drivers" describes the function of the software drivers delivered with the core.
- Chapter 14, "Quick Start Example Design"Chapter 3, "Quick Start Example Design" provides instructions to quickly generate the core and run the example design through implementation and simulation using the default settings.
- Chapter 15, "Detailed Example Design (Standard Format)" provides detailed information about the core when generated in the standard CORE Generator format, including a description of files and the directory structure generated
- Chapter 16, "Detailed Example Design (EDK format)" provides detailed information about the core when generated in the Standard Embedded Development Kit (EDK) format, including a description of files and the directory structure generated.
- Appendix A, "RTC Time Stamp Accuracy" describe the necessity of accurate time stamps, essential to the Precise Timing Protocol across the network link, and provides some of the ways inaccuracies are introduced.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays. Signal names in text also.	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File →Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *User Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Dark Shading	Items that are not supported or reserved	This feature is not supported
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** `[`*`option_name`*`]` *`design_name`*

Convention	Meaning or Use	Example
Braces { }	A list of items from which you must choose one or more	`lowpwr ={on|off}`
Vertical bar |	Separates items in a list of choices	`lowpwr ={on|off}`
Angle brackets < >	User-defined variable or in code samples	<directory name>

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active low	`usr_teof_n` is active low.

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Guide Contents" for details. See "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-5 FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

List of Abbreviations

The following table describes acronyms used in this manual.

Acronym	Spelled Out
AV	Audio Video
AVB	Audio Video Bridging
BMCA	Best Master Clock Algorithm
CRC	Cyclic Redundancy Check
DA	Destination Address
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDK	Embedded Development Kit
EMAC	Ethernet MAC
FCS	Frame Check Sequence
FIFO	First In First Out
FPGA	Field Programmable Gate Array.
Gbps	Gigabits per second
GMII	Gigabit Media Independent Interface
GUI	Graphical User Interface
HDL	Hardware Description Language
IES	Incisive Unified Simulator
I/F	Interface
IO	Input/Output
IP	Intellectual Property
ISE®	Integrated Software Environment
KHz	Kilo Hertz
LLDP	Link Layer Discovery Protocol
MAC	Media Access Controller
Mbps	Megabits per second
MDIO	Management Data Input/Output
MHS	Microprocessor Hardware Description: a proprietary file format, using the .mhs file extension, for a XPS project
MHz	Mega Hertz
ms	milliseconds
MPMC	Multi-Port Memory Controller
ns	nanoseconds

Acronym	Spelled Out
PHY	physical-side interface
PHYAD	Physical Address
PLB	Processor Local Bus
PTP	Precise Timing Protocol
REGAD	Register Address
RTC	Real Time Clock
RO	Read Only
R/W	Read/Write
Rx	Receive
SFD	Start of Frame Delimiter
SRP	Stream Reservation Protocol
TEMAC	Tri-Mode Ethernet MAC
TCP/IP	Transmission Control Protocol / Internet Protocol.
TOE	TCP/IP Offload Engine
Tx	Transmitter
UCF	User Constraints File
us	microseconds
VHDL	VHSIC Hardware Description Language (VHSIC an acronym for Very High-Speed Integrated Circuits)
VLAN	Virtual LAN (Local Area Network)
WO	Write Only
XCO	Xilinx CORE Generator core source file
XPS	Xilinx Platform Studio (part of the EDK software)
XPS_LL_TEMAC	XPS LocalLink Tri-Mode Ethernet MAC

XILINX®

Chapter 1

Introduction

This chapter introduces the core and provides related information including recommended design experience, additional resources, technical support, and how to submit feedback to Xilinx.

The Ethernet AVB Endpoint core is a fully verified solution that supports Verilog-HDL and VHDL. In addition, the example design in this guide is provided in both Verilog and VHDL formats.

System Requirements

Windows

- Windows XP Professional 32-bit/64-bit
- Windows Vista Business 32-bit/64-bit Linux
- Red Hat Enterprise Linux WS v4.0 32-bit/64-bit
- Red Hat Enterprise Desktop v5.0 32-bit/64-bit (with Workstation Option)
- SUSE Linux Enterprise (SLE) desktop and server v10.1 32-bit/64-bit

Software

- ISE® software v12.2

About the Core

The Ethernet AVB Endpoint core is available through the Xilinx CORE Generator™ software included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, see the Ethernet AVB Endpoint product page. For information about licensing options, see Chapter 2, "Licensing the Core."

Recommended Design Experience

Although the Ethernet AVB Endpoint core is a fully verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, previous experience building high-performance, pipelined FPGA designs using Xilinx implementation software and user constraint files (UCFs) is recommended. In addition, previous experience using the Embedded Development Kit (EDK) and developing embedded software applications is recommended. Contact your local Xilinx representative for a closer review and estimation for your specific requirements.

Additional Core Resources

For detailed information and updates about the Ethernet AVB Endpoint core, see the following documents, available from the product page.

- Ethernet AVB Endpoint *Data Sheet*
- Ethernet AVB Endpoint *User Guide*

From the document directory after generating the core:

- Ethernet AVB Endpoint *Release Notes*

Technical Support

For technical support, see www.support.xilinx.com/. Questions are routed to a team of engineers with expertise using the Ethernet AVB Endpoint core.

Xilinx provides technical support for use of this product as described in this guide. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the Ethernet AVB Endpoint core and the documentation supplied with the core.

Ethernet AVB Endpoint Core

For comments or suggestions about the Ethernet AVB Endpoint core, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm/

Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about this document, submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm/

Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

XILINX®

Chapter 2

Licensing the Core

This chapter provides instructions for obtaining a license key for the Ethernet AVB Endpoint core, which you must do before using the core in your designs. The Ethernet AVB Endpoint core is provided under the terms of the Xilinx Core Site License Agreement.

Before you Begin

This chapter assumes that you have installed the required Xilinx® ISE® Design Suite version following the instructions provided by the Xilinx ISE Installation, Licensing and Release Notes Guide, www.xilinx.com/support/documentation/dt_ise.htm. Detailed software requirements can be found on the product web page for this core, www.xilinx.com/products/ipcenter/DO-DI-EAVB-EPT.htm.

License Options

The Ethernet AVB Endpoint core provides three licensing options. After installing the required ISE Design Suite version, choose a license option.

Simulation Only

The Simulation Only Evaluation license key is provided with the ISE CORE Generator tool. This key lets you assess core functionality with either the example design provided with the Ethernet AVB Endpoint core, or alongside your own design and allows you to demonstrate the various interfaces to the core in simulation. (Functional simulation is supported by a dynamically generated HDL structural model.)

Full System Hardware Evaluation

The Full System Hardware Evaluation license key is available at no cost and lets you fully integrate the core into an FPGA design, place and route the design, evaluate timing, and perform back-annotated gate-level simulation of the core using the demonstration test bench provided with the core.

In addition, the license key lets you generate a bitstream from the placed and routed design, which can then be downloaded to a supported device and tested in hardware. The core can be tested in the target device for a limited time before *timing out* (ceasing to function), at which time it can be reactivated by reconfiguring the device.

Full

The Full license key is available when you purchase a license for the core and provides full access to all core functionality both in simulation and in hardware, including:

- Functional simulation support
- Back annotated gate-level simulation support
- Full implementation support including place and route and bitstream generation
- Full functionality in the programmed device with no time outs

Obtaining Your License Key

This section contains information about obtaining a simulation, full system hardware, and full license keys.

Simulation License

No action is required to obtain the Simulation Only Evaluation license key; it is provided by default with the Xilinx CORE Generator software.

Full System Hardware Evaluation License

To obtain a Full System Hardware Evaluation license, do the following:

1. Navigate to the product page for this core.
2. Click **Evaluate**.
3. Follow the instructions to install the required Xilinx ISE software and IP Service Packs.

Obtaining a Full License Key

To obtain a Full license key, please follow these instructions:

1. Purchase the license through your local sales office. Once the order has been entered, an email will be sent to your Account Administrator with instructions on how to access the account.
2. Navigate to the product page for this core: www.xilinx.com/products/ipcenter/DO-DI-EAVB-EPT.htm
3. Click **Order**.
4. Follow the instructions to generate the required license key on the Xilinx Product Licensing Site, www.xilinx.com/getproduct.

Further details can be found at www.xilinx.com/products/ipcenter/ipaccess_fee.htm.

Installing the License File

The Simulation Only Evaluation license key is provided with the ISE software CORE Generator system and does not require installation of an additional license file. For the Full System Hardware Evaluation license and the Full license, an email will be sent to you containing instructions for installing your license file. Additional details about IP license key installation can be found in the ISE Design Suite Installation, Licensing and Release Notes document.

Chapter 3

Overview of Ethernet Audio Video Bridging

Figure 3-1 illustrates a potential home network, consisting of wired (ethernet) and wireless components, which utilize the technology being defined by the IEEE802.1 Audio Video Bridging Task Group. This illustrates potential audio/video *talkers* (for example, a Cable or Satellite Content Provider, or home MP3 player) and a number of potential *listeners* (for example TV sets which may exist in several rooms). In addition, users of the various household PCs may be surfing the internet. It is important to note that all of this data is being transferred across the single home network backbone.

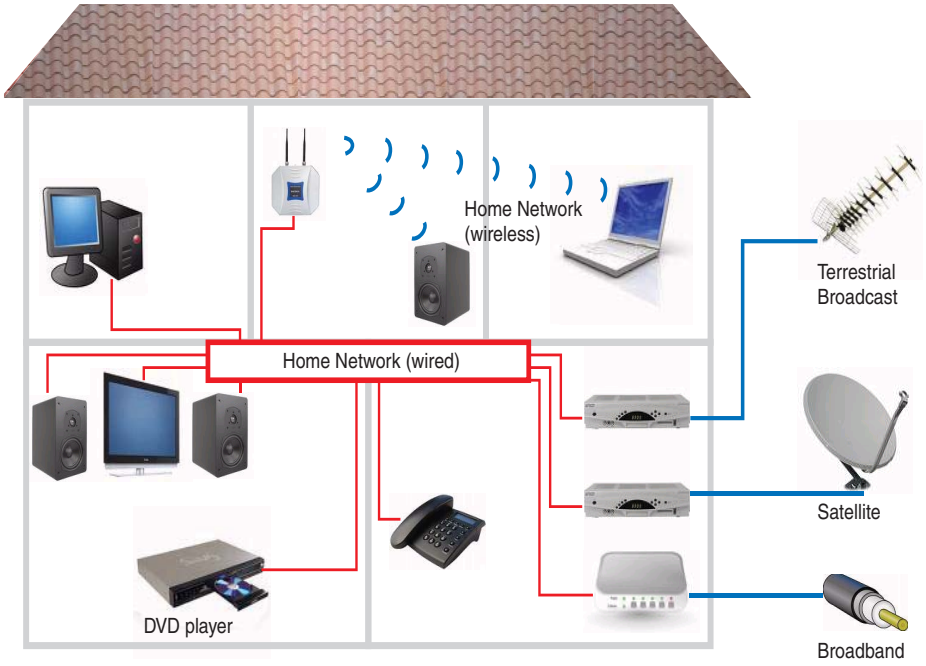

Figure 3-1: **Example AVB Home Network**

To understand the requirements of this network, we must differentiate between certain types of data:

- **Audio and Video streaming data**, referred to in this document as *AV traffic*. Requires a good quality of service to avoid, for example, TV picture breakup, and must be transferred reliably and with guaranteed low latency.
- **Other data**, referred to in this document as *legacy traffic*. Does not have the strict requirement of AV traffic: data can be started, stopped and delayed without serious consequence for example, a PC surfing the internet.

For these reasons, an important aspect of the AVB technology is therefore to prioritize the audio/video streaming data (AV traffic) over that of standard data transfer (legacy traffic).

AVB Specifications

The IEEE802.1 Audio Video Task Group is currently working on new specifications which combine to define this technology:

P802.1AS

This specification defines how to synchronize a common time base across an entire AVB network, utilizing functionality from IEEE1588 (version 2), and known as Precise Timing Protocol (PTP). This common time base is in the form of a Real Time Clock (RTC), effectively a large counter which consists of a 32-bit nanoseconds field and a 48-bit seconds field. A single device on the network is designated as the clock master (by automatic resolution) using a Best Master Clock Algorithm (BMCA). All other devices resolve to be slaves. Using the *P802.1AS PTP*, all slave devices will regularly update their own RTC to match that of the network clock master.

This common time base has various applications:

- It can be used to synchronize media clocks (audio clocks or video pixel clocks) across the entire network to match audio and video data rates between talkers and listeners.
- It can be used by an Ethernet AVB Endpoint System, that is, configured as a "talker", to time a class measurement interval for an SR stream. (The class measurement interval for a stream depends upon the SR class associated with the stream: SR class A corresponds to a class measurement interval of 125 microseconds; SR class B corresponds to a class measurement interval of 250 microseconds). The class measurement interval for a stream is used to limit the number of data frames that are placed into the stream's queue per class measurement interval.
- It can be used by higher layer applications (for example *IEEE1722*) to provide presentation time stamps for audio and video data. This is used, for example, to synchronize the *lip sync* on a TV set so a viewer hears the words at the same time as they see the lips move.

The *P802.1AS* specification is implemented in the Ethernet AVB Endpoint using a combination of hardware and software. The hardware components are incorporated into the core, and the software component is provided with the core in the form of drivers. These drivers should be run on an embedded processor (MicroBlaze™ or PowerPC®).

P802.1Qav

This specification defines the mechanism for queuing and forwarding AV traffic from a talker to a listener across the network. This can involve several network hops (network bridge devices that the data must pass through).

P802.1Qav is also responsible for enforcing the 75% maximum bandwidth restriction across each link of the network that can be reserved for the AV traffic.

Only a subset of the *P802.1Qav* requirements for an Endpoint is implemented in the Ethernet AVB Endpoint core, with the following assumptions for *talkers* and *listeners*:

Talker Assumptions

- *AV traffic* Ethernet frames that are input to the Ethernet AVB Endpoint use the VLAN priority values that the Bridges in the network recognize as being associated with SR classes exclusively for transmitting stream data.
- *Legacy traffic* Ethernet frames that are input to the Ethernet AVB Endpoint do not use the VLAN priority values that the Bridges in the network recognize as being associated with SR classes exclusively for transmitting stream data.
- The credit shaping algorithm operates on the *AV traffic* port; so in order to comply with the transmission selection rules for *P802.1Qav,* all Ethernet frames input on the *AV traffic* port are assumed to be of the same SR Class. However, the Ethernet AVB Endpoint does not enforce this rule and it is acceptable to send a mix of SR Class A and SR Class B Ethernet frames on the *AV traffic* port. In this case the Ethernet AVB Endpoint will not prioritize SR Class A Ethernet frames over SR Class B Ethernet frames; instead it will apply the credit-based shaper algorithm to all of the Ethernet frames that are input on the *AV traffic* port.
- The Ethernet AVB Endpoint assumes that any per-stream traffic management has been done prior to *AV traffic* being input on the *AV traffic* port. To comply with the transmission selection rules for *P802.1Qav* it is assumed that if multiple streams are input to the Ethernet AVB Endpoint via the *AV traffic* port, that the credit-based shaper algorithm has been used per stream as the transmission selection mechanism, prior to the *AV traffic* being input on the AV traffic port.
- If multiple AV streams are input to the Ethernet AVB Endpoint via the AV traffic port, it is assumed that the IdleSlope/SendSlope control registers (See "Tx Arbiter Send Slope Control Register" and "Tx Arbiter Idle Slope Control Register") are programmed correctly to be the sum of the IdleSlope /SendSlope values for all the streams that are input on the AV traffic port. The credit-based shaper algorithm used on the *AV traffic* port will enforce a hiLimit/loLimit on the credits to ensure that this interface is not misused.

Listener Assumptions

- The Ethernet AVB Endpoint provides a mechanism for identifying received *AV traffic* for either one or two SR classes (see "Rx Filtering Control Register"); however, it does not provide any buffering for AV traffic Ethernet frames. Buffering is expected to be done outside the Ethernet AVB Endpoint, after it has separated out the AV traffic Ethernet frames, as the buffering requirements are expected to be application-specific.

P802.1Qat

This specification defines a Stream Reservation Protocol (SRP) which must be used over the AVB network. Every listener that intends to receive audio/video AV traffic from a talker must make a request to reserve that bandwidth. Both the talker and every bridge device that exists between the talker and the listener has the right to decline this request. Only if each device is capable of routing the new AV traffic stream without violating the 75% total bandwidth restriction (when taking into account previously granted bandwidth commitments), will the bandwidth request be successful. However, after granted, this audio / video stream is reliably routed across the network until the reservation is removed.

Note: No hardware components are required for the *P802.1Qat* specification because this is a pure software task. This software is not provided by the Ethernet AVB Endpoint core.

Typical Implementation

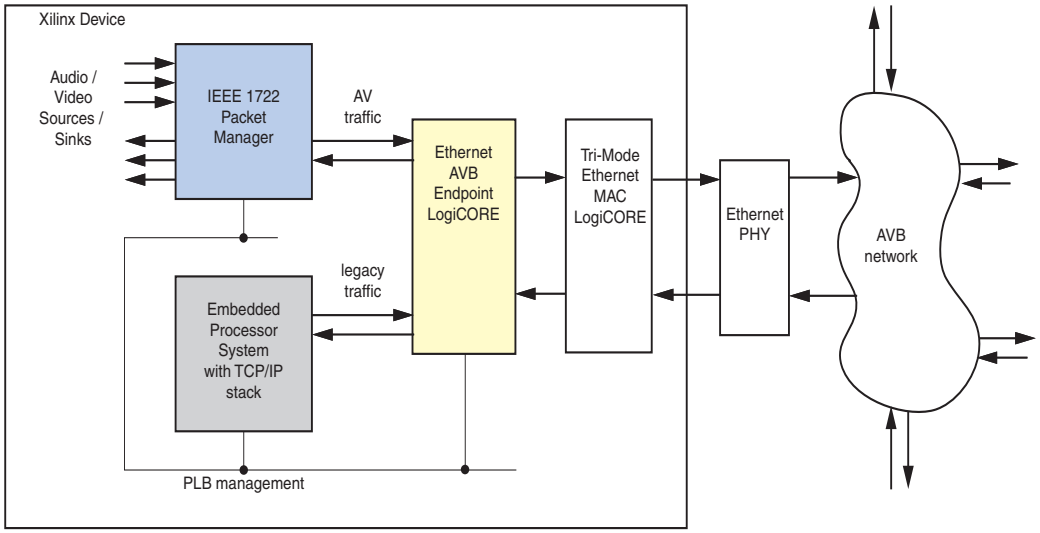

Figure 3-2: **Example Ethernet AVB Endpoint System**

Figure 3-2 illustrates a typical implementation for the Ethernet AVB Endpoint core. Endpoint refers to a talker or listener device from the example network shown in Figure 3-1, as opposed to an intermediate bridge function, which is not supported.

In the implementation, the Ethernet AVB Endpoint core is shown connected to a Xilinx Tri-Mode Ethernet MAC core, which in turn is connected to an AVB capable network. All devices attached to this network should be AVB capable to obtain the full Quality of Service advantages for the AV traffic. This AVB network can be a professional or consumer network (as illustrated in Figure 3-1).

Figure 3-2 illustrates that the Ethernet AVB Endpoint core supports the two main types of data interfaces at the client side:

1. The **AV traffic** interface is intended for the Quality of Service audio/video data. Illustrated are a number of audio/video sources (for example, a DVD player), and a number of audio/video sinks (for example, a TV set). The Ethernet AVB Endpoint gives priority to the **AV traffic** interface over the **legacy traffic** interface, as dictated by *IEEE P802.1Qa*v 75% bandwidth restrictions.
2. The **legacy traffic** interface is maintained for *best effort* ethernet data: Ethernet as we know it today (for example, the PC surfing the internet in Figure 3-1). Wherever possible, priority is given to the **AV traffic** interface (as dictated by *IEEE P802.1Qav* bandwidth restrictions) but a minimum of 25% of the total Ethernet bandwidth is always available for legacy ethernet applications.

The **AV traffic** interface in Figure 3-2 is shown as interfacing to a 1722 Packet Manager block. The *IEEE1722* is also an evolving standard which will specify the embedding of audio/video data streams into Ethernet Packets. The 1722 headers within these packets can optionally include presentation time stamp information. Contact Xilinx for further system-level information.

Chapter 4

Generating the Core

The Ethernet AVB Endpoint core is fully configurable using the CORE Generator™ software, which provides a Graphical User Interface (GUI) for defining parameters and options. For help starting and using the CORE Generator software, see the documentation supplied with the ISE® software, including the *CORE Generator User Guide*, available from www.xilinx.com/support/software_manuals.htm.

Ethernet AVB GUI Page 1

Figure 4-1 shows page 1 of the Ethernet AVB Endpoint GUI customization screen.

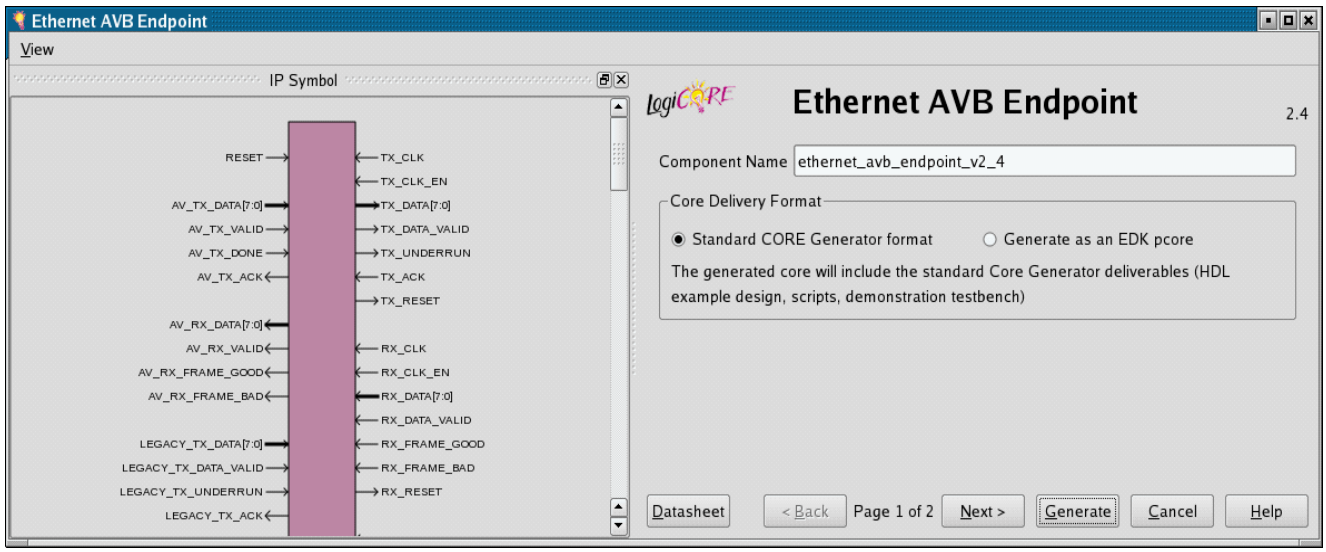

Figure 4-1: **GUI Page 1**

Component Name

The component name is used as the base name of the output files generated for the core. Names must begin with a letter and must be composed from the following characters: a through z, 0 through 9 and "_".

Core Delivery Format

The Ethernet AVB Endpoint core can be delivered in two different formats, selectable from this section of the CORE Generator software Customization GUI:

- Standard CORE Generator software format (provided for the standard ISE software environment)

  This option will deliver the core in the standard CORE Generator software output format, as used by many other cores including previous versions of this core and all other Ethernet LogiCORE™ IP solutions.

  When generated in this format, the core is designed to interface to the LogiCORE IP Tri-Mode Ethernet MAC or the LogiCORE IP Embedded Tri-Mode Ethernet MAC wrappers (available in selected Virtex® families). See Chapter 12, "System Integration".

  When generated in this format, "Ethernet AVB GUI Page 2" is available for customization of the "PLB Interface".

- Generate as an EDK pcore (provided for the Embedded Development Kit)

  This option will deliver the core in the standard pcore format, suitable for directly importing into the Xilinx Embedded Development Kit (EDK) environment.

  When generated in this format, the core is designed to interface to the XPS LocalLink Tri-Mode Ethernet MAC (xps_ll_temac). See Chapter 12, "System Integration".

  When generated in this format, page 2 of the GUI is not available; the "PLB Interface" will be configured dynamically by the EDK Xilinx Platform Studio (XPS) software.

For directory and file definitions for the two available formats, see Chapter 15, "Detailed Example Design (Standard Format)" and Chapter 16, "Detailed Example Design (EDK format)."

Ethernet AVB GUI Page 2

Figure 4-2 shows page 2 of the Ethernet AVB Endpoint GUI customization screen. This page provides options for configuring the "PLB Interface" of the core. This option is only required when generating in the Standard CORE Generator software format.

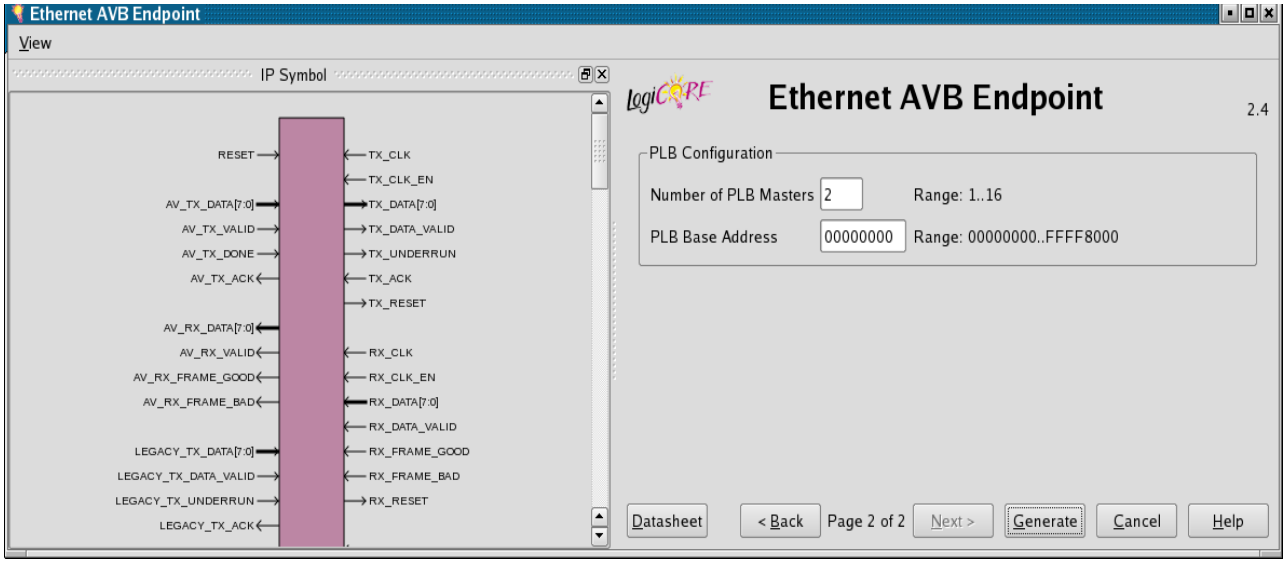

Figure 4-2: **GUI Page 2**

Number of PLB Masters

The Ethernet AVB Endpoint core is a PLB slave. On the connected PLB, there may be several PLB Masters. Each slave must uniquely acknowledge individual masters using unique PLB signals during transactions. For this reason, set this integer value to match the number of PLB masters that will be present on the PLB.

PLB Base Address

The Ethernet AVB Endpoint core is a PLB slave. The base address of the core must be selected. Valid range is 0x00000000 to 0xFFFF8000. The least significant 15 bits of the base address must be set to 0 (bits 17 to 31 of the PLB Base Address).

Parameter Values in the XCO File

XCO file parameter names and their values are identical to the names and values shown in the GUI.

Table 4-1 shows the XCO file parameters and values and summarizes the GUI defaults. The following is an example of the CSET parameters in an XCO file:

```
CSET component_name=eth_avb_endpoint_v2_4
CSET number_of_plb_masters=2
CSET plb_base_address=00000000
```

Table 4-1: **XCO File Values and Default Values**

Parameter	XCO File Values	Default GUI Setting
component_name	ASCII text starting with a letter and based on the following character set: a..z, 0..9 and _	eth_avb_endpoint_v2_4
generate_as_edk_pcore	Select between true and false	false
number_of_plb_masters	Select from the range: 1 to 16	2
plb_base_address	Select from the range: 0x00000000 to 0xFFFF8000	0x00000000

Output Generation

The output files generated by the CORE Generator software are placed in the project directory. The list of output files includes the following items.

- The netlist file for the core
- Supporting CORE Generator software files
- Release notes and documentation
- Subdirectories containing an HDL example design
- Scripts to run the core through the back-end tools and to simulate the core using Mentor Graphics ModelSim v6.5c, Cadence Incisive Enterprise Simulator (IES) v9.2, and Synopsys VCS and VCS MX 2009.12

See the following chapters for a complete description of the CORE Generator software output files and for detailed information about the HDL example design.

- Chapter 14, “Quick Start Example Design”
- Chapter 15, “Detailed Example Design (Standard Format)”
- Chapter 16, “Detailed Example Design (EDK format)”

XILINX®

Chapter 5

Core Architecture

As described in Chapter 4, "Generating the Core", the core can be generated in one of two formats, the functionality of which is described in this chapter:

- "Standard CORE Generator Format" (provided for the standard ISE® software environment)

  This option will deliver the core in the standard CORE Generator™ output format, as used by many other cores including previous versions of this core and all other Ethernet LogiCORE™ IP solutions.

  When generated in this format, the core is designed to interface to the LogiCORE IP Tri-Mode Ethernet MAC or the LogiCORE IP Embedded Tri-Mode Ethernet MAC wrappers (available in selected Virtex® families). See Figure 5-1.

- "EDK pcore Format"(provided for the Embedded Development Kit)

  This option will deliver the core in the standard pcore format, suitable for directly importing into the Xilinx Embedded Development Kit (EDK) environment.

  When generated in this format, the core is designed to interface to the XPS LocalLink Tri-Mode Ethernet MAC (xps_ll_temac). See Figure 5-2.

Standard CORE Generator Format

Figure 5-1 illustrates the functional blocks of the Ethernet AVB Endpoint core when it is generated in standard CORE Generator format. As illustrated, this is intended to be connected to the LogiCORE IP Tri-Mode Ethernet MAC (or to the LogiCORE IP Embedded Ethernet Wrappers available in certain Virtex devices).

Each of the functional blocks illustrated will be introduced in the following sections of this chapter. However, observe from the figure that:

- The Host I/F (management interface) of the Tri-Mode Ethernet MAC is connected directly to the Ethernet AVB Endpoint LogiCORE IP. This enables the MAC to be fully configured via the "PLB Interface" of the Ethernet AVB Endpoint core.
- The core provides two independent full-duplex interfaces for customer logic: the "AV Traffic Interface" and the "Legacy Traffic Interface".
- The "Legacy Traffic Interface" contains "MAC Header Filters"; these are provided to replace the Address Filter functionality of the LogiCORE IP Tri-Mode Ethernet MACs (which must be disabled).

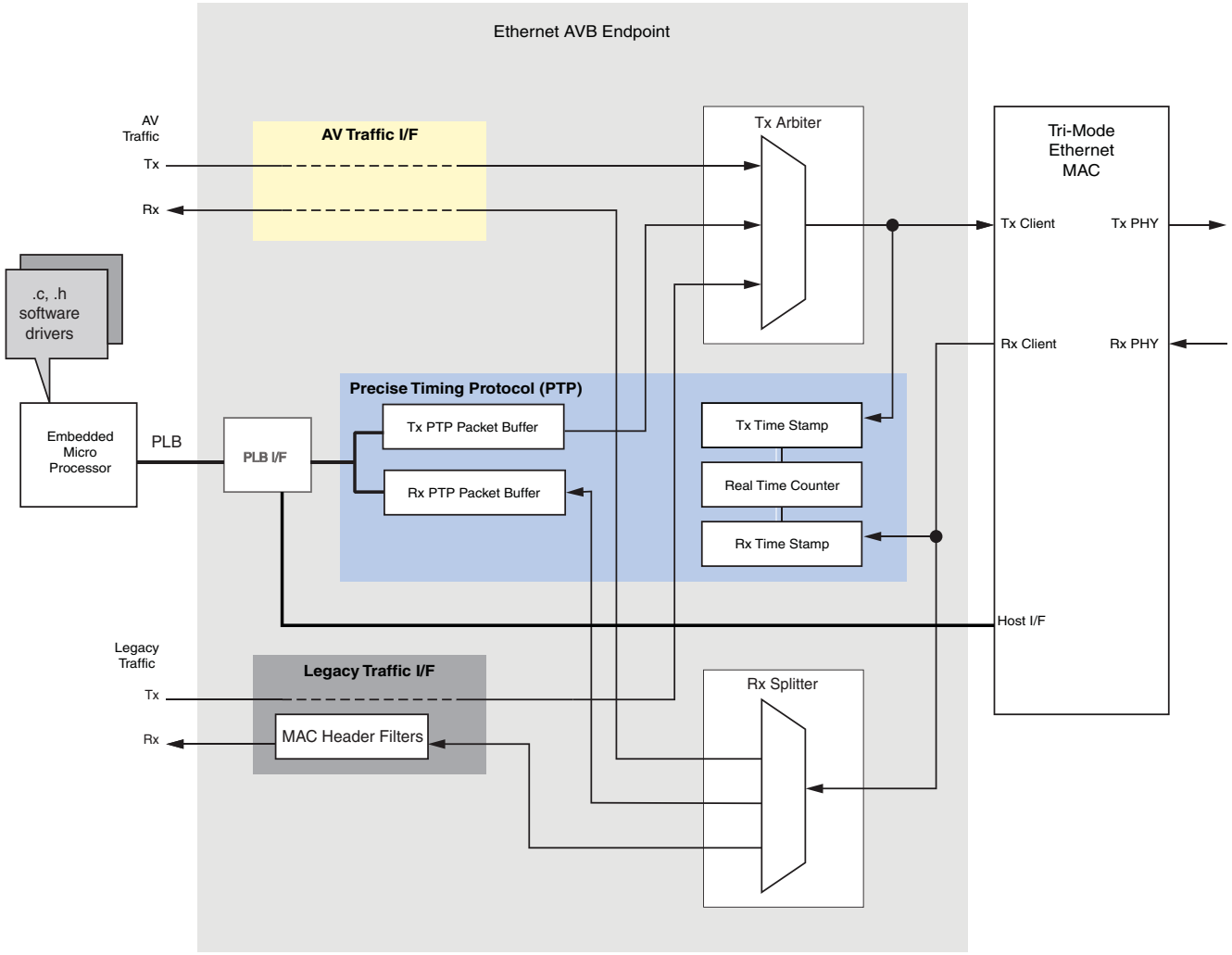

Figure 5-1: **Ethernet AVB Endpoint Core Block Diagram for Connection to LogiCORE IP Tri-Mode Ethernet MAC**

EDK pcore Format

Figure 5-2 illustrates the functional blocks of the Ethernet AVB Endpoint core when it is generated in EDK pcore format. As illustrated, this is intended to be connected to the XPS LocalLink Tri-Mode Ethernet MAC.

Each of the functional blocks illustrated will be introduced in the following sections of this chapter. However, observe from the figure that:

- The xps_ll_temac contains its own PLB interface. Consequently, the logic connecting the "PLB Interface" of the Ethernet AVB Endpoint core to the Host I/F (as seen in Figure 5-1) is not present in this case.
- The "Legacy Traffic Interface" of the Ethernet AVB Endpoint core is connected directly to the xps_ll_temac; this allows the xps_ll_temac core to source and sink legacy frame data, such as TCP/IP protocol traffic. The full duplex "AV Traffic Interface" remains for connection to custom logic.
- The "MAC Header Filters", as seen in Figure 5-1, are not present in this case. The xps_ll_temac instead contains its own Address Filter logic.

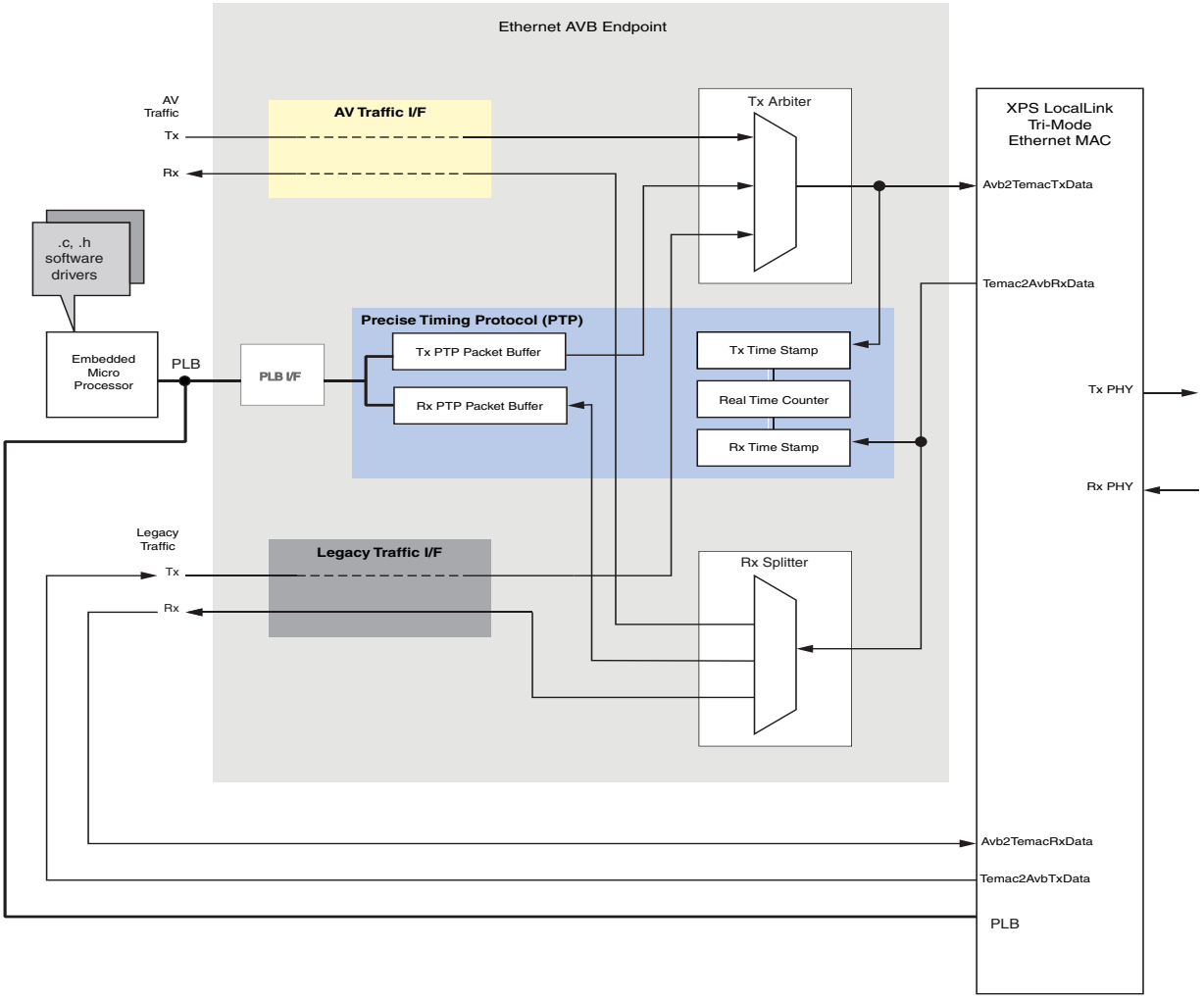

Figure 5-2: **Ethernet AVB Endpoint Core Block Diagram for Connection to the XPS Tri-Mode Ethernet MAC (xps_ll_temac) in the EDK**

Functional Block Description

The following functional blocks described in the following sections are illustrated in Figure 5-1 and Figure 5-2.

PLB Interface

The core provides a PLB version 4.6 interface as its configuration port to provide easy integration with the Xilinx Embedded Development Kit and access to an embedded processor (MicroBlaze™ or PowerPC®), which is required to run the "Software Drivers." All the configuration and status register address space of the Ethernet AVB Endpoint core can be accessed through the PLB.

Additionally, when the core is generated in "Standard CORE Generator Format", the PLB logic provides a logic shim which is connected to the Host I/F of the supported Xilinx Tri-Mode MAC core; this enables all configuration and status registers of the MAC to also be available via the PLB. See Chapter 10, "Configuration and Status" for more information.

AV Traffic Interface

The AV traffic interface provides a dedicated full duplex port for the high priority AV data. See Chapter 6, "Ethernet AVB Endpoint Transmission," and Chapter 7, "Ethernet AVB Endpoint Reception" for further information.

Legacy Traffic Interface

The legacy traffic interface provides a dedicated full-duplex port for the legacy data, as described in Chapter 6, "Ethernet AVB Endpoint Transmission," and Chapter 7, "Ethernet AVB Endpoint Reception".

When the core is generated in "Standard CORE Generator Format" then "Legacy MAC Header Filters" are provided on the receiver path. These filters have a greater flexibility than the address filter provided in the LogiCORE Tri-Mode Ethernet MACs (which must be disabled).

When the core is generated in "EDK pcore Format", the legacy traffic interface is designed to connect directly to the ports of the xps_ll_temac core; please see Figure 5-2 and Chapter 12, "System Integration". Additionally, the "Legacy MAC Header Filters" are not included since the xps_ll_temac can optionally contain its own Address Filter logic.

Tx Arbiter

Data for transmission over an AVB network can be obtained from three types of sources:

1. **AV Traffic.** For transmission from the AV Traffic I/F of the core.
2. **Precise Timing Protocol (PTP) Packets**. Initiated by the software drivers using the dedicated hardware "Tx PTP Packet Buffers."
3. **Legacy Traffic**. For transmission from the Legacy Traffic I/F of the core.

The transmitter (Tx) arbiter must prioritize these packets. To aid with this, the arbiter contains configuration registers that can be used to set the percentage of available Ethernet bandwidth reserved for AV traffic. To comply with the specifications, this should not be configured to exceed 75%. The arbiter then polices this bandwidth restriction for the AV traffic and ensures that on average, it is never exceeded. Consequently, despite the AV traffic having a higher priority than the legacy traffic, there is always remaining bandwidth available to schedule legacy traffic. The output of the arbiter should be connected directly to the client Tx interface of the connected Ethernet MAC, as illustrated. See Chapter 6, "Ethernet AVB Endpoint Transmission," for further information.

Rx Splitter

The input to the splitter is connected directly to the client Receive (Rx) interface of the connected Ethernet MAC. Received data from an AVB network can be of three types:

- **Precise Timing Protocol (PTP) Packets**. Routed to the dedicated hardware "Rx PTP Packet Buffers" which can be accessed by the "Software Drivers." PTP packets are identified by searching for a specific value in the MAC Length/Type field.
- **AV Traffic**. Routed to the AV Traffic I/F of the core. These packets are identified by searching for MAC packets containing a MAC VLAN field with one of two possible configurable VLAN priority values; the VLAN priorities are defaulted to values of 3 and 2.
- **Legacy Traffic:**. Routed to the Legacy Traffic I/F of the core. All packet types which are not identified as PTP or AV Traffic will be considered legacy traffic.

See Chapter 7 for further information.

MAC Header Filters

The MAC Header Filters provided on the receiver legacy traffic path when the core is generated in "Standard CORE Generator Format". These filters provide a greater flexibility than the standard address filter provided in the LogiCORE IP Tri-Mode Ethernet MACs (which must be disabled). The MAC Header Filters include the ability to filter across any of the initial 16-bytes of an Ethernet frame, including the ability to filter only on the Destination Address, Length/Type Field, VLAN tag (if present), or any bit-wise match combination of the preceding. Eight individual MAC Header Filters are provided, each of which is separately configured. See Chapter 7, "Ethernet AVB Endpoint Reception" for further information.

When the core is generated in "EDK pcore Format", the "Legacy MAC Header Filters" are not included since the xps_ll_temac can optionally contain its own Address Filter logic.

Precise Timing Protocol Blocks

The various hardware Precise Timing Protocol (PTP) blocks within the core provide the dedicated hardware to implement the *IEEE P802.1AS* specification. However, the full functionality is only achieved using a combination of these hardware blocks coupled with functions provided by the "Software Drivers" (run on an embedded processor). Consequently the following hardware block descriptions also give some insight into the software driver functionality.

Note: The following definitions provide only a simplistic concept of PTP protocol operation. For detailed information about the PTP protocol, see the *IEEE P802.1AS* specification.

Tx PTP Packet Buffers

The PTP packet buffer contains pre-initialized templates for seven different PTP packets defined by the *P802.1AS* specification. The buffer contents are read/writable through the PLB and a separate configuration register within the core requests to the Tx Arbiter which of these seven packets is to be transmitted. A dedicated interrupt signal will be generated by the core whenever a PTP packet has been transmitted.

The software drivers provided with the core, using the PLB and dedicated interrupts, will use this interface to periodically update specific fields within the PTP packets, and request transmission of these packets. See Chapter 9, "Precise Timing Protocol Packet Buffers" for further information.

Tx Time Stamp

Whenever a PTP packet is transmitted, a sample of the current nanosecond value of the local RTC is taken. This timestamp value is written into a dedicated field within the Tx PTP Packet Buffer, where it is accessible along side the content of the PTP frame that was just transmitted. By the time the Tx PTP buffer raises its dedicated interrupt, this time stamp is available for the microprocessor to read. This sampling of the RTC is performed in hardware for accuracy. See Chapter 9, "Precise Timing Protocol Packet Buffers" for further information.

Rx PTP Packet Buffers

Received PTP Packets will be written to the Rx PTP Packet Buffer by the Rx Splitter. This buffer is capable of storing up to 16 separate PTP frames. Whenever a PTP packet is received, a dedicated interrupt will be generated. The contents of the stored packets can be read via the PLB. The oldest stored frame will always be overwritten by a new frame reception and so a configuration register within the core will contain a pointer to the most recently stored packet.

The software drivers provided with the core, using the PLB and dedicated interrupt, will use this interface to decode, and then act on, the received PTP packet information. See Chapter 9, "Precise Timing Protocol Packet Buffers" for further information.

Rx Time Stamp

When a PTP packet is received, a sample of the current nanosecond value of the RTC is taken. This timestamp value is written into a dedicated field within the Rx PTP Packet Buffer, where it is accessible along side the PTP frame that was just received. By the time the Rx PTP buffer raises its dedicated interrupt, this time stamp is available for the microprocessor to read. This sampling of the RTC is performed in hardware for accuracy. See Chapter 9, "Precise Timing Protocol Packet Buffers" for further information.

RTC

A significant component of the PTP network wide timing synchronization mechanism is the Real Time Counter (RTC), which provides the common time of the network. Every device on the network will maintain its own local version.

The RTC is effectively a large counter which consists of a 32-bit nanosecond field (the unit of this field is 1 nanosecond and this field will count the duration of exactly one second, then reset back to zero) and a 48-bit second field (the unit of this field is one second: this field will increment when the nanosecond field saturates at 1 second). The seconds field will only wrap around when its count fully saturates. The entire RTC is therefore designed never to wrap around in our lifetime. The RTC counter is implemented as part of the core in hardware.

Conceptually, this counter is not related to the frequency of the clock used to increment it. A configuration register within the core provides a configurable increment rate for this counter; this increment register simply takes the value of the clock period which is being used to increment the RTC. However, the resolution of this increment register is very fine, in units of 1/1048576 ($1/2^{20}$) fraction of one nanosecond. For this reason, the RTC increment rate can be adjusted to a very fine degree of accuracy. This provides the following features:

- The RTC can be incremented from any available clock frequency that is greater than the AVB standards defined minimum of 25 MHz. However, the faster the frequency of the clock, the smaller will be the step increment and the smoother will be the overall RTC increment rate. Xilinx recommends clocking the RTC logic at 125 MHz because this is a readily available clock source (obtained from the transmit clock source of the Ethernet MAC at 1 Gbps speed). This frequency significantly exceeds the minimum performance of the *P802.1AS* specification.
- When acting as a clock slave, the rate adjustment of the RTC can be matched to that of the network clock master to an exceptional level of accuracy. The software drivers provided with this core will periodically calculate the increment rate error between itself and the master and update the RTC increment value accordingly.

The core also contains a configuration register which allows a large step change to be made to the RTC. This can be used to initialize the RTC, after power-up. It is also used to make periodic corrections, as required, by the software drivers when operating as a clock slave; if the increment rates are closely matched, these periodic step corrections will be small. See Chapter 9, “Precise Timing Protocol Packet Buffers” for further information.

Software Drivers

Software Drivers are delivered with the Ethernet AVB Endpoint core. These drivers provide functions which utilize the dedicated hardware within the core for the PTP *IEEE P802.1AS* specification. Functions include:

- The Best Master Clock Algorithm (BMCA) to determine whether the core should operate in master clock or slave clock mode
- PTP Clock Master functions
- PTP Clock Slave functions (which accurately synchronize the local Real Time Clock (RTC) to match that of the network clock master)

If the core is acting as clock master, then the software drivers delivered with the core will periodically sample the current value of the RTC and transmit this value to every device on the network using the *P802.1* defined PTP packets. The hardware "Tx Time Stamp" logic, using the mechanism defined in *P802.1AS*, ensures the accuracy of this RTC sample mechanism.

If the core is acting as a clock slave, then the local RTC will be closely matched to the value and frequency of the network clock master. This is achieved, in part, by receiving the PTP frames transmitted across the network by the clock master (and containing the masters sampled RTC value). The PTP mechanism will also track the total routing delay across the network between the clock master and itself. The software drivers use this data, in conjunction with recent historical data, to calculate the error between its local RTC counter and that of the RTC clock master. The software will then periodically calculate an RTC correction value and an updated increment rate, and these values are written to appropriate RTC configuration registers. See Chapter 13, "Software Drivers" for further information.

Tri-Mode Ethernet MACs

Although not part of the Ethernet AVB Endpoint core, a Xilinx Tri-Mode Ethernet MAC core is a requirement of the system (see Figure 5-1 and Figure 5-2). The IEEE Audio Video Bridging technology stipulates the following configuration requirements on this MAC:

- The MAC must only operate in full-duplex mode
- The MAC must only operate at 100 Mbps and/or 1 Gbps
- VLAN mode must be enabled (the AV traffic will always contain VLAN fields)
- Flow Control is not supported on the network and must be disabled
- Jumbo Frames are not supported and must be disabled
- The built-in Address Filter Module of the MAC must be disabled

Core Interfaces

All ports of the core are internal connections in FPGA fabric.

All clock signals are inputs and no clock resources are used by the core. This enables clock circuitry to be implemented externally to the core netlist, providing full flexibility for clock sharing with other custom logic.

Clocks and Reset

Table 5-1 defines the clock and reset signals which are required by the Ethernet AVB Endpoint core.

Table 5-1: **Clocks and Resets**

Signal	Direction	Description
reset	Input	Asynchronous reset for the entire core
rtc_clk	Input	Reference clock used to increment the "RTC." The minimum frequency is 25 MHz. Xilinx recommends a 125 MHz clock source.
tx_clk	Input	The MAC transmitter clock, provided by the Tri-Mode Ethernet MAC.
tx_clk_en	Input	A clock enable signal: this must be used as a qualifier for `tx_clk`.
rx_clk	Input	The MAC receiver clock, provided by the Tri-Mode Ethernet MAC.
rx_clk_en	Input	A clock enable signal: this must be used as a qualifier for `rx_clk`.
host_clk	Input	An input clock for the management interface of the connected Tri-Mode Ethernet MAC. This clock can be independent, or could be shared with `PLB_clk`. This signal is only present when the core is generated in "Standard CORE Generator Format".
PLB_clk	Input	The input clock reference for the PLB bus.
tx_reset	Output	Output reset signal for logic on the Legacy Traffic and AV Traffic transmitter paths. This reset signal is synchronous to `tx_clk`; the reset is asserted when a transmitter path reset request is made to the "Software Reset Register."
rx_reset	Output	Output reset signal for logic on the Legacy Traffic and AV Traffic receiver paths. This reset signal is synchronous to `rx_clk`; the reset is asserted when a receiver path reset request is made to the "Software Reset Register."

Legacy Traffic Interface

Legacy Traffic Transmitter Path Signals

Table 5-2 defines the core client-side legacy traffic transmitter signals. These signals are used to transmit data from the legacy client logic into the core. All signals are synchronous to the MAC transmitter clock, `tx_clk`, which must be qualified by the corresponding clock enable, `tx_clk_en` (see "Clocks and Resets").

Table 5-2: **Legacy Traffic Signals: Transmitter Path**

Signal	Direction	Description
legacy_tx_data[7:0]	Input	Frame data to be transmitted is supplied on this port
legacy_tx_data_valid	Input	A data valid control signal for data on the `legacy_tx_data[7:0]` port
legacy_tx_underrun	Input	Asserted by the client to force the MAC to corrupt the current frame
legacy_tx_ack	Output	Handshaking signal asserted when the current data on `legacy_tx_data[7:0]` has been accepted.

Legacy Traffic Receiver Path Signals

Table 5-3 defines the core client side legacy traffic receiver signals. These signals are used by the core to transfer data to the client. All signals are synchronous to the MAC receiver clock, `rx_clk`, which must be qualified by the corresponding clock enable, `rx_clk_en` (see "Clocks and Resets").

Table 5-3: **Legacy Traffic Signals: Receiver Path**

Signal	Direction	Description
legacy_rx_data[7:0]	Output	Legacy frame data received is supplied on this port.
legacy_rx_data_valid	Output	Control signal for the `legacy_rx_data[7:0]` port

Table 5-3: **Legacy Traffic Signals: Receiver Path**

Signal	Direction	Description
legacy_rx_frame_good	Output	Asserted at the end of frame reception to indicate that the frame should be processed by the MAC client.
legacy_rx_frame_bad	Output	Asserted at the end of frame reception to indicate that the frame should be discarded by the MAC client: either the frame contained an error, or it was intended for the PTP or AV traffic channel.
legacy_rx_filter_match[7:0]	Output	This output is only present when the "MAC Header Filters" are present (when the core is generated in "Standard CORE Generator Format"). When present, each bit in the bus corresponds to one of the unique "Legacy MAC Header Filters." A bit is asserted, in alignment with `legacy_rx_data_valid` signal, if the corresponding filter number obtained a match.

AV Traffic Interface

AV Traffic Transmitter Path Signals

Table 5-4 defines the core client-side AV traffic transmitter signals, used to transmit data from the AV client logic into the core. All signals are synchronous to the MAC transmitter clock, `tx_clk`, which must be qualified by the corresponding clock enable, `tx_clk_en` (see "Clocks and Resets").

Table 5-4: **AV Traffic Signals: Transmitter Path**

Signal	Direction	Description
av_tx_data[7:0]	Input	Frame data to be transmitted is supplied on this port
av_tx_valid	Input	A data valid control signal for data on the `av_tx_data[7:0]` port
av_tx_done	Input	Asserted by the AV client to indicate that further frames, following the current frame, are/are not held in a queue.
av_tx_ack	Output	Handshaking signal asserted when the current data on `av_tx_data[7:0]` has been accepted.

AV Traffic Receiver Path Signals

Table 5-5 defines the core client side AV traffic receiver signals, used by the core to transfer data to the AV client. All signals are synchronous to the MAC receiver clock, `rx_clk`, which must be qualified by the corresponding clock enable, `rx_clk_en` (see "Clocks and Resets").

Table 5-5: **AV Traffic Signals: Receiver Path**

Signal	Direction	Description
av_rx_data[7:0]	Output	AV frame data received is supplied on this port.
av_rx_valid	Output	Control signal for the `av_rx_data[7:0]` port
av_rx_frame_good	Output	Asserted at the end of frame reception to indicate that the frame should be processed by the MAC client.
av_rx_frame_bad	Output	Asserted at the end of frame reception to indicate that the frame should be discarded by the MAC client: either the frame contained an error, or it was intended for the PTP or legacy traffic channel.

Tri-Mode Ethernet MAC Client Interface

Table 5-6, Table 5-7 and Table 5-8 list the ports of the core which connect directly to the port signals of the Tri-Mode Ethernet MAC core, which are identically named. For detailed information about the Tri-Mode Ethernet MAC ports, see the Tri-Mode Ethernet MAC User Guide (UG138).

MAC Transmitter Interface

These signals connect directly to the identically named Tri-Mode Ethernet MAC signals and are synchronous to `tx_clk`.

Table 5-6: **Tri-Mode Ethernet MAC Transmitter Interface**

Signal	Direction	Description
tx_data[7:0]	Output	Frame data to be transmitted is supplied on this port
tx_data_valid	Output	A data valid control signal for data on the `tx_data[7:0]` port
tx_underrun	Output	Asserted to force the MAC to corrupt the current frame
tx_ack	Input	Handshaking signal asserted when the current data on `tx_data[7:0]` has been accepted by the MAC.

MAC Receiver Interface

These signals connect directly to the identically named Tri-Mode Ethernet MAC signals and are synchronous to `rx_clk`

Table 5-7: **Tri-Mode Ethernet MAC Receiver Interface**

Signal	Direction	Description
rx_data[7:0]	Input	Frame data received is supplied on this port.
rx_data_valid	Input	Control signal for the `rx_data[7:0]` port
rx_frame_good	Input	Asserted at the end of frame reception to indicate that the frame should be processed by the Ethernet AVB Endpoint core.
rx_frame_bad	Input	Asserted at the end of frame reception to indicate that the frame should be discarded by the MAC client.

MAC Management Interface

This interface is only present when the core is generated in "Standard CORE Generator Format", designed for connection to LogiCORE IP Tri-Mode Ethernet MAC devices.

When present, these signals connect directly to the identically named LogiCORE IP Tri-Mode Ethernet MAC signals (except where stated in Table 5-8) and are synchronous to `host_clk`. When present, all MAC configuration and MDIO register space is address mapped into the PLB of the Ethernet AVB Endpoint core. A logic shim automatically drives this interface to access the MAC when the appropriate PLB address space is accessed.

Table 5-8: **Tri-Mode Ethernet MAC Host Interface (Configuration/Status)**

Signal	Direction	Description
host_opcode[1:0]	Output	Defines the MAC operation (configuration or MDIO, read or write)
host_addr[9:0]	Output	Address of the MAC register to access
host_wr_data[31:0]	Output	Data to be written to the MAC register
host_rd_data_mac[31:0]	Input	Data read from the MAC register (connect to the host_rd_data[31:0] signal of the MAC)
host_rd_data_stats[31:0]	Input	Data read from the Ethernet Statistics core (connect to the host_rd_data[31:0] signal of the Ethernet Statistics core, if present). If the statistics core is not used, then connect to logic 0.
host_miim_sel	Output	When asserted, the MAC will access the MDIO port, when not asserted, the MAC will access configuration registers
host_req	Output	Used to initiate a transaction onto the MDIO

Table 5-8: **Tri-Mode Ethernet MAC Host Interface (Configuration/Status)**

Signal	Direction	Description
host_miim_rdy	Input	When high, the MAC has completed its MDIO transaction
host_stats_lsw_rdy	Input	Signal provided by the Ethernet Statistics core to indicate that the lower 32-bits of the statistic counter value is present on the host_rd_data_stats[31:0] port. If the statistics core is not used, then connect to logic 0.
host_stats_msw_rdy	Input	Signal provided by the Ethernet Statistics core to indicate that the upper 32-bits of the statistic counter value is present on the host_rd_data_stats[31:0] port. If the statistics core is not used, then connect to logic 0.

Processor Local Bus (PLB) Interface

The Processor Local Bus (PLB) on the Ethernet Audio Video core is designed to be integrated directly in the Xilinx Embedded Development Kit (EDK) where it can be easily integrated and connected to the supported embedded processors (MicroBlaze or PowerPC). As a result, the PLB interface does not require in-depth understanding, and the following information is provided for reference only. See the EDK documentation for further information.

The PLB interface, defined by IBM, can be complex and support many usage modes (such as multiple bus masters). It can support single or burst read/writes, and can support different bus widths and different peripheral bus widths.

The general philosophy of the Ethernet AVB Endpoint core has been to implement a PLB interface which is as simple as possible. The following features are provided:

- 32-bit data width.
- Implements a simple PLB slave.
- Supports single read/writes only (no burst or page modes).

PLB Interface

Table 5-9 defines the signals on the PLB bus. For detailed information, see the IBM PLB specification. Shaded rows represent signals not used by this core; inputs are ignored and outputs are tied to a constant. These signals are synchronous to `PLB_clk`; see "Clocks and Resets" for additional information.

Table 5-9: **PLB Signals**

PIN Name	Direction	Description
PLB_clk	Input	Reference clock for the PLB
PLB_reset	Input	Reset for the PLB, synchronous to PLB_clk
PLB_ABus[0:31]	input	PLB address bus
PLB_UABus[0:31]	Input	PLB upper address bus
PLB_PAvaild	Input	PLB primary address valid indicator
PLB_SAValid	Input	Unused. PLB secondary address valid indicator.
PLB_rdPrim	Input	Unused. PLB secondary to primary read request indicator.
PLB_wrPrim	Input	Unused. PLB secondary to primary write request indicator.
PLB_masterID [0:log2(NUM_MASTERS)]	Input	PLB current master identifier
PLB_abort	Input	PLB abort request indicator
PLB_busLock	Input	Unused. PLB bus lock.
PLB_RNW	Input	PLB read not write
PLB_BE[0:3]	Input	PLB byte enables
PLB_MSize[0:1]	Input	PLB master data bus size
PLB_size[0:3]	Input	PLB transfer size. Only support size 0.
PLB_type[0:2]	Input	PLB transfer type. Only support type 0.
PLB_TAttribute[0:15]	Input	Unused. PLB transfer attribute bus.
PLB_lockErr	Input	Unused. PLB lock error indicator.
PLB_wrDBus[0:31]	Input	PLB write data bus
PLB_wrBurst	Input	PLB write burst transfer indicator.
PLB_rdBurst	Input	PLB read burst transfer indicator.
PLB_rdPendReq	Input	Unused. PLB pending read request priority.
PLB_wrPendReq	Input	Unused. PLB pending write request priority.
PLB_rdPendPri[0:1]	Input	Unused. PLB pending read bus request indicator.

Table 5-9: **PLB Signals *(Cont'd)***

PIN Name	Direction	Description
PLB_wrPendPri[0:1]	Input	Unused. PLB pending read bus request indicator.
PLB_reqPri[0:1]	Input	Unused. PLB request priority.
Sl_addrAck	Output	Slave address acknowledge
Sl_SSize[0:1]	Output	Slave data bus size.
Sl_wait	Output	Slave wait indicator.
Sl_rearbitrate	Output	Slave rearbitrate bus indicator. Not used, tied to logic 0.
Sl_wrDack	Output	Slave write data acknowledge
Sl_wrComp	Output	Slave write transfer complete indicator
Sl_WrBTerm	Output	Slave terminate write burst transfer.
Sl_rdBus[0:31]	Output	Slave read data bus
Sl_rdWdAddr[0:3]	Output	Slave read word address
Sl_rdDAck	Output	Slave read data acknowledge
Sl_rdComp	Output	Slave read transfer complete indicator
Sl_rdBTerm	Output	Slave terminate read burst transfer.
Sl_MBusy[0:NUM_MASTERS-1]	Output	Slave busy indicator
Sl_MWrErr[0:NUM_MASTERS-1]	Output	Unused, tied to logic 0. Slave write error indicator.
Sl_MRdErr[0:NUM_MASTERS-1]	Output	Unused, tied to logic 0. Slave read error indicator.
Sl_MIRQ[0:NUM_MASTERS-1]	Output	Unused, tied to logic 0. Slave interrupt indicator.

Interrupt Signals

Table 5-10 defines the interrupt signals asserted by the core. All interrupts are active high and are automatically asserted. All interrupts, required by the "Software Drivers" delivered with the core, are cleared by software access to an associated configuration register. It is recommended that these interrupts are routed to the input of an EDK Interrupt Controller module as part of the embedded processor subsystem.

Table 5-10: **Interrupt Signals**

Signal	Direction	Description
interrupt_ptp_timer	Output	This interrupt is asserted every 1/128 second as measured by the "RTC." This acts as a timer for the PTP software algorithms.
interrupt_ptp_tx	Output	This is asserted following the transmission of any PTP packet from the "Tx PTP Packet Buffers."
interrupt_ptp_rx	Output	This is asserted following the reception of any PTP packet into the "Rx PTP Packet Buffers."

PTP Signals

Table 5-11 defines the signals which are output from the core by the "Precise Timing Protocol Blocks." These signals are provided for reference only and may be used by an application. For example, the 1722 Packet Managers, as illustrated in Figure 3-2, require the following:

- `clk8k`: this marks the class measurement interval to be used for traffic shaping for SR class A AV traffic.
- `rtc_nanosec_field` and `rtc_sec_field`: used in the 1722 presentation time stamp logic.

Table 5-11: **PTP Signals**

Signal	Direction	Description
rtc_nanosec_field[31:0]	Output	This is the synchronized nanoseconds field from the "RTC."
rtc_sec_field[47:0]	Output	This is the synchronized seconds field from the "RTC."
clk8k	Output	This is an 8KHz clock which is derived from, and synchronized in frequency, to the "RTC."
rtc_nanosec_field_1722[31:0]	Output	The IEEE1722 specification contains a different format for the "RTC,"provided here as an extra port. This is derived and is in sync with the IEEE802.1 AS RTC. If desired, this port can be used as the RTC reference for 1722 Packet Manager blocks, as illustrated in Figure 3-2. See also "IEEE1722 Real Time Clock Format," page 81.

XILINX®

Chapter 6

Ethernet AVB Endpoint Transmission

As illustrated in Figure 5-1, data for transmission over an AVB network can be obtained from three types of sources:

1. **AV Traffic.** For transmission from the "Tx AV Traffic I/F" of the core.
2. **Precise Timing Protocol (PTP) Packets**. Initiated by the software drivers using the dedicated hardware "Tx PTP Packet Buffer."
3. **Legacy Traffic**. For transmission from the "Tx Legacy Traffic I/F" of the core.

Tx Legacy Traffic I/F

The signals forming the Tx Legacy Traffic I/F are defined in Table 5-2. All signals are synchronous to the Tri-Mode Ethernet MAC transmitter clock, `tx_clk`, which must always be qualified by the corresponding clock enable, `tx_clk_en` (see Table 5-1).

This interface is intentionally *identical* to the client transmitter interface of the supported Xilinx Tri-Mode Ethernet MAC core (there is a one-to-one correspondence between signal names of the *block*-level wrapper from the Tri-Mode Ethernet MAC example design, after the `legacy_` prefix is removed). This provides backwards compatibility–all existing MAC client-side designs can connect to the legacy Ethernet port unmodified.

Error Free Legacy Frame Transmission

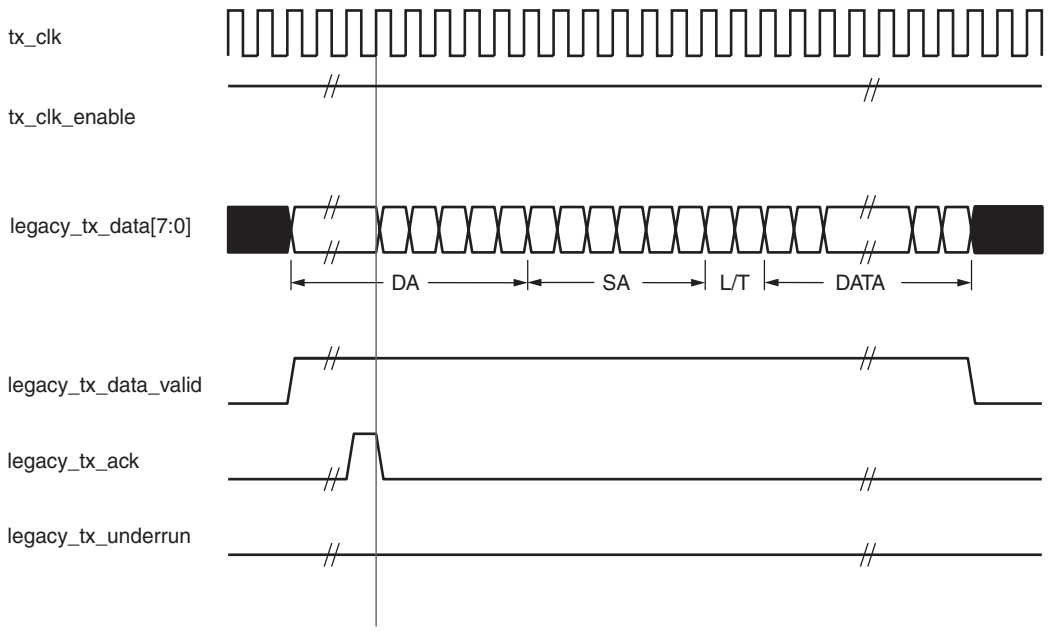

Figure 6-1: **Normal Frame Transmission across the Legacy Traffic Interface**

Figure 6-1 illustrates the timing of a normal frame transfer. When the legacy client initiates a frame transmission, it places the first column of data onto the `legacy_tx_data[7:0]` port and asserts a logic 1 onto `legacy_tx_data_valid`. After the Ethernet AVB Endpoint core reads the first byte of data, it asserts the `legacy_tx_ack` signal. On the next and subsequent rising clock edges, the client must provide the remainder of the data for the frame. The end of frame is signalled to the core by taking the `legacy_tx_data_valid` to logic 0.

Errored Legacy Frame Transmission

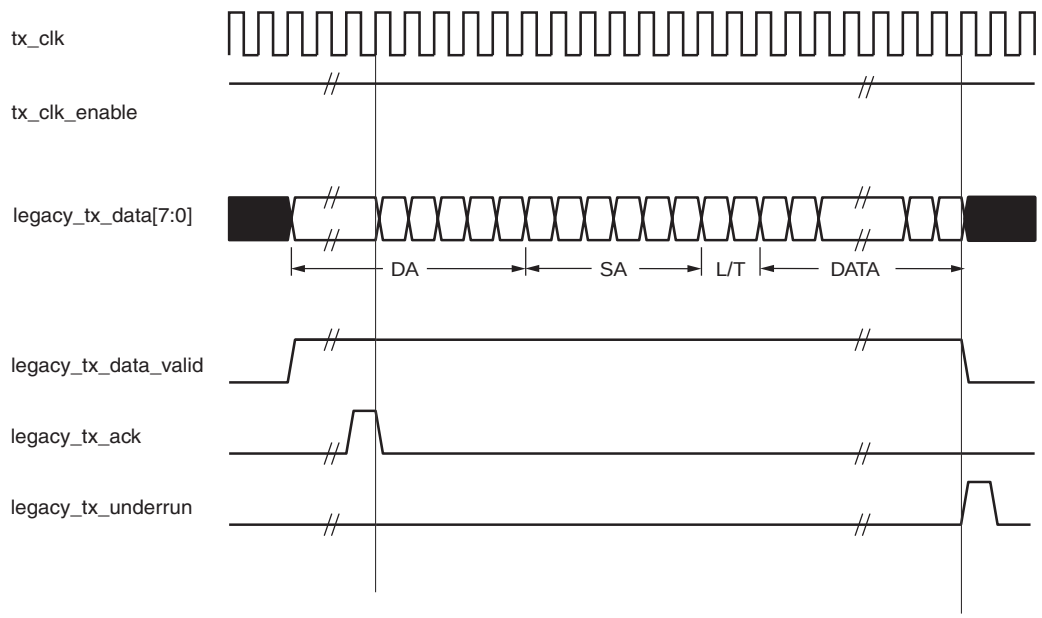

Figure 6-2: **Legacy Frame Transmission with Underrun**

The `legacy_tx_underrun` is provided to give full backwards compatibility between the Legacy Traffic I/F and the client interface of the Tri-Mode Ethernet MAC. The `legacy_tx_underrun` provides a mechanism to inject an error into a frame before transmission is completed. This can occur, for example, if a FIFO connected to the Legacy client empties during transmission.

To error the frame, the `legacy_tx_underrun` signal may be asserted during the data transmission or up to 1 valid clock cycle after `legacy_tx_data_valid` goes low.

Tx AV Traffic I/F

The signals forming the Tx AV Traffic I/F are defined in Table 5-4. All signals are synchronous to the Tri-Mode Ethernet MAC transmitter clock, `tx_clk`, which must always be qualified by the corresponding clock enable, `tx_clk_en` (see Table 5-1). See ("Talker Assumptions," page 31) for information about the expectations for the AV traffic input to the Ethernet AVB Endpoint on this interface.

This interface is intentionally very similar to the "Tx Legacy Traffic I/F." Note, however, that the legacy traffic does not contain a signal that is equivalent to `av_tx_done`. Additionally, the AV does not contain a signal that is equivalent to `legacy_tx_underrun`: no mechanism is currently provided on the AV interface to signal an error in a frame which is currently undergoing transmission.

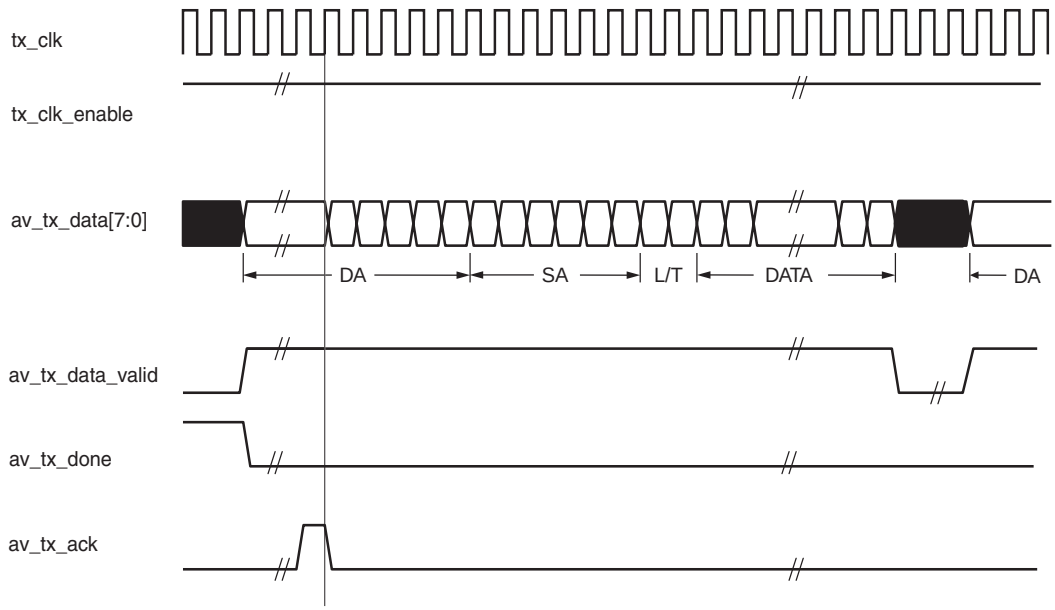

Figure 6-3: **Normal Frame Transmission across the AV Traffic Interface**

Figure 6-3 illustrates the timing of a normal frame transfer. When the AV client initiates a frame transmission, it places the first column of data onto the `av_tx_data[7:0]` port and asserts a logic 1 onto `av_tx_valid`.

After the Ethernet AVB Endpoint core reads the first byte of data, it asserts the `av_tx_ack` signal. On the next and subsequent rising clock edges, the client must provide the remainder of the data for the frame. The end of frame is signalled to the core by taking the `av_tx_valid` to logic 0.

In Figure 6-3, following the end of frame transmission, the `av_tx_done` signal is held low, which indicates to the "Tx Arbiter" that another AV frame is queued. Unless the configurable bandwidth restrictions have been exceeded, this *parks* the "Tx Arbiter" onto the AV traffic queue. Figure 6-3 then illustrates the client asserting the `av_tx_valid` signal to request a subsequent frame, and the frame transmission cycle of Figure 6-3 repeats. However, if no further AV traffic frames are queued, the `av_tx_done` signal should be set to logic 1 immediately following the end of frame transmission. This than allows the "Tx Arbiter" to schedule legacy traffic transmission (if any legacy frames are queued).

If, following the end of frame reception, the bandwidth allocation for AV traffic has been exceeded, the "Tx Arbiter" switches to service the legacy traffic regardless of the state of the `av_tx_done` signal.

For this reason, the `av_tx_done` signal should be considered an aid to the "Tx Arbiter" to help make best use of the available network bandwidth. Asserting this signal after all AV traffic has been serviced immediately allows the "Tx Arbiter" to service the legacy traffic. This helps achieve in excess of the 25% minimum allocation for the legacy traffic. However, holding off the assertion of `av_tx_done` will not act as cheat mode to exceed the maximum bandwidth allocation for the AV traffic.

Tx Arbiter

Overview

As illustrated in Figure 5-1, data for transmission over an AVB network can be obtained from three types of sources:

1. **AV Traffic.** For transmission from the AV Traffic I/F of the core.
2. **Precise Timing Protocol (PTP) Packets**. Initiated by the software drivers using the dedicated hardware "Tx PTP Packet Buffer."
3. **Legacy Traffic**. For transmission from the Legacy Traffic I/F of the core.

The transmitter (Tx) arbiter selects from these three sources in the following manner.

- If there is AV packet available and the programmed AV bandwidth limitation is not exceeded, then the AV packet is transmitted
- otherwise the Tx arbiter checks to see if there are any PTP packets to be transmitted
- otherwise if there is an available legacy packet then this will be transmitted.

The Ethernet AVB Endpoint core contains configuration registers to set up the percentage of available Ethernet bandwidth reserved for AV traffic. To comply with the IEEE P802.1 Qav specification these should not be configured to exceed 75%. The arbiter then polices this bandwidth restriction for the AV traffic and ensures that on average, it is never exceeded. Consequently, despite the AV traffic having a higher priority than the legacy traffic, there is always remaining bandwidth available to schedule legacy traffic.

The relevant configuration registers for programming the bandwidth percentage dedicated to AV traffic are defined in Chapter 10, "Configuration and Status" and are:

- "Tx Arbiter Send Slope Control Register"
- "Tx Arbiter Idle Slope Control Register"

These registers are defaulted to values which dedicate **up to** 75% of the overall bandwidth to the AV traffic. This is the maximum legal percentage that will be defined in the *IEEE802.1* AVB standards.

In many implementations, it may be unnecessary to change these register values. Correct use of the `av_tx_done` signal, as defined in "Tx AV Traffic I/F," will allow the Tx Arbiter to share the bandwidth allocation efficiently between the AV and Legacy sources (even in the situations where the AV traffic requires less than 75% of the overall bandwidth).

However, for the cases that require less than 75% of the overall bandwidth, careful configuration can result in a *smoother* (less bursty) transmission of the AV traffic, which should prevent frame *bunching* across the AVB network.

Credit Based Traffic Shaping Algorithm

To enforce the bandwidth policing of the AV Traffic, a credit-based shaper algorithm has been implemented in the Ethernet AVB Endpoint core. Figure 6-4 illustrates the basic operation of the algorithm and indicates how the Tx Arbiter decides which Ethernet frame to transmit.

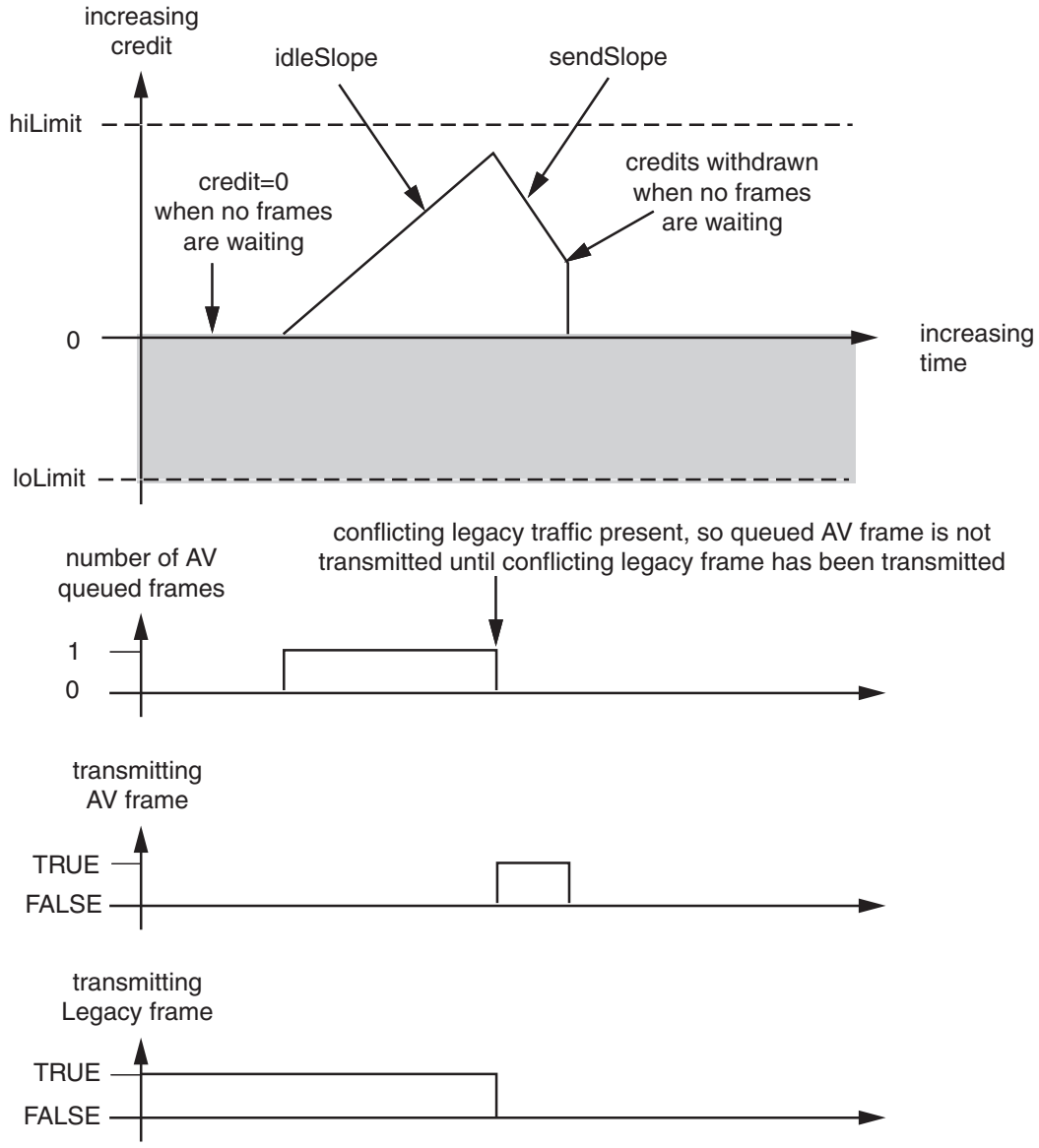

Figure 6-4: **Credit-based Shaper Operation**

Figure 6-4 illustrates the key features of the credit based algorithm, which are:

- The Tx Arbiter will schedule queued transmission from the "Tx AV Traffic I/F" if the algorithm is in credit (greater or equal to 0).
- If there is less than 0 credit (not shown in Figure 6-4, but the credit can sink below 0), then the Tx Arbiter will not allow AV traffic to be transmitted; legacy traffic, if queued, will be scheduled instead.
- When no AV traffic is queued, any positive credit will be lost and the credit is reset to 0.
- When AV traffic is queued, and until the time at which the Tx Arbiter is able to schedule it (while waiting for an in-progress legacy frame to complete transmission), credit can be gained at a rate defined by the **idleSlope.**

- During AV traffic transmission, credit is removed at a rate defined by the **sendSlope.**
- The **hiLimit** and **loLimit** settings impose a fixed range on the possible values of credit. If the available credit hits one of these limits, it will not exceed, but saturate at the magnitude of that limit. These limits are fixed in the netlist to ensure that the interface is not used incorrectly.

The overall intention of the two settings **idleSlope** and **sendSlope** is to spread out the AV traffic transmission as evenly as possible over time, preventing periods of bursty AV transmission surrounded by idle AV transmission periods. No further background information is provided in this document with regard to the credit-based algorithm.

The remainder of this section describes the **idleSlope**, and **sendSlope** variables from the perspective of the Ethernet AVB Endpoint core.

Tx Arbiter Bandwidth Control

The Ethernet AVB Endpoint core contains four configuration registers, used for setting the cores local definitions of "idleSlope," and "sendSlope."

The configuration register settings are described in general, and then from the point of view of a single example which describes the calculations made to set the register default values. This example dedicates up to 75% of the overall bandwidth to be reserved for the AV traffic (leaving at least 25% for the Legacy Traffic).

The calculations described are independent of Ethernet operating speed (no re-calculation is required when changing between Ethernet speeds of 1 Gbps and 100 Mbps).

idleSlope

The general equation is:

```
idleSlopeValue=(AV percentage / 100) x 8192
```

In this example, dedicating up to 75% of the total bandwidth to the AV traffic, we obtain:

```
idleSlopeValue=(75 / 100) x 8192 = 6144
```

The calculated value for the `idleSlopeValue` should be written directly to the "Tx Arbiter Idle Slope Control Register." This provides a per-byte increment value when relating this to Legacy Ethernet frame transmission.

sendSlope

The general equation is:

```
sendSlopeValue=((100 - AV percentage) / 100) x 8192
```

In this example, dedicating up to 75% of the total bandwidth to the AV traffic, we obtain:

```
sendSlopeValue=((100 - 75) / 100) x 8192 = 2048
```

The calculated value for the `sendSlopeValue` should be written directly to the "Tx Arbiter Send Slope Control Register." This provides a per-byte decrement value when relating this to AV Ethernet frame transmission.

hiLimit

The general equation is:

```
hiLimitValue = 2000 x idleSlopeValue
```

In this general equation, the value of 2000 is obtained from the maximum number of bytes which may be present in legacy frames (an *Envelope* frame as defined in *IEEE802.3* can be of size 2000 bytes).

In this example, dedicating up to 75% of the total bandwidth to the AV traffic, we obtain:

```
hiLimitValue = 2000 x 6144 = 12288000
```

loLimit

The general equation is:

```
loLimitValue = 1518 x sendSlopeValue
```

In this general equation, the value of 1518 is obtained from the maximum number of bytes which may be present in AV frames.

In this example, dedicating up to 75% of the total bandwidth to the AV traffic, we obtain:

```
loLimitValue = 1518 x 2048 = 3108864
```

XILINX®

Chapter 7

Ethernet AVB Endpoint Reception

Rx Splitter

The input to the Rx splitter (see Figure 5-1) is connected directly to the client Receive (Rx) interface of the connected Ethernet MAC. Received data from an AVB network can be of three types:

- **Precise Timing Protocol (PTP) Packets**. Routed to the dedicated hardware "Rx PTP Packet Buffer" which can be accessed by the "Software Drivers." PTP packets are identified by searching for a specific MAC Destination Address.
- **AV Traffic**. Routed to the "Rx AV Traffic I/F"of the core. These packets are identified by searching for MAC packets containing a MAC VLAN field with one of two possible configurable VLAN priority values (see "Rx Filtering Control Register").
- **Legacy Traffic:**. Routed to the"Rx Legacy Traffic I/F" of the core. All packet types which are not identified as PTP or AV Traffic will be considered legacy traffic.

Rx Legacy Traffic I/F

The signals forming the Rx Legacy Traffic I/F are defined in Table 5-3. All signals are synchronous to the Tri-Mode Ethernet MAC receiver clock, `rx_clk`, which must always be qualified by the corresponding clock enable, `rx_clk_en` (see Table 5-1).

This interface is intentionally *identical* to the client receiver interface of the supported Xilinx Tri-Mode Ethernet MAC core (there is a one-to-one correspondence between signal names of the *block*-level wrapper from the Tri-Mode Ethernet MAC example design, after the `legacy_` prefix is removed). This provides backward compatibility–all existing MAC client-side designs which use the clock enable should be able to connect to the legacy Ethernet port unmodified.

Operation of the Rx Legacy Traffic Interface is closely connected with the frame header match results of the "Legacy MAC Header Filters." If the filters are enabled and do not obtain a match, the frame data does not appear on this interface (`legacy_rx_data_valid` and `legacy_rx_frame_good/legacy_rx_frame_bad` are not asserted). When a match is obtained these signals are asserted as described in the following sections.

Error Free Legacy Frame Reception

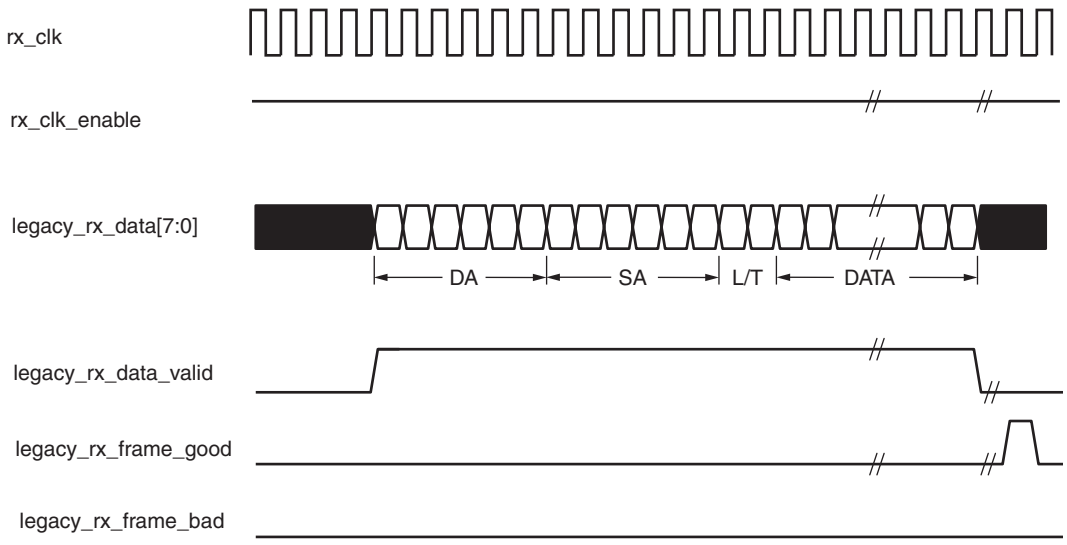

Figure 7-1: **Normal Frame Reception across the Legacy Traffic Interface**

Figure 7-1 illustrates the timing of a normal inbound error free frame transfer that has been accepted by the "Legacy MAC Header Filters" The legacy client must be prepared to accept data at any time; there is no buffering within the core to allow for latency in the receive client. After frame reception begins, data is transferred on consecutive clock enabled cycles to the receive client until the frame is complete. The core asserts the `legacy_rx_frame_good` signal to indicate that the frame was intended for the legacy traffic client and was successfully received without error.

Errored Legacy Frame Reception

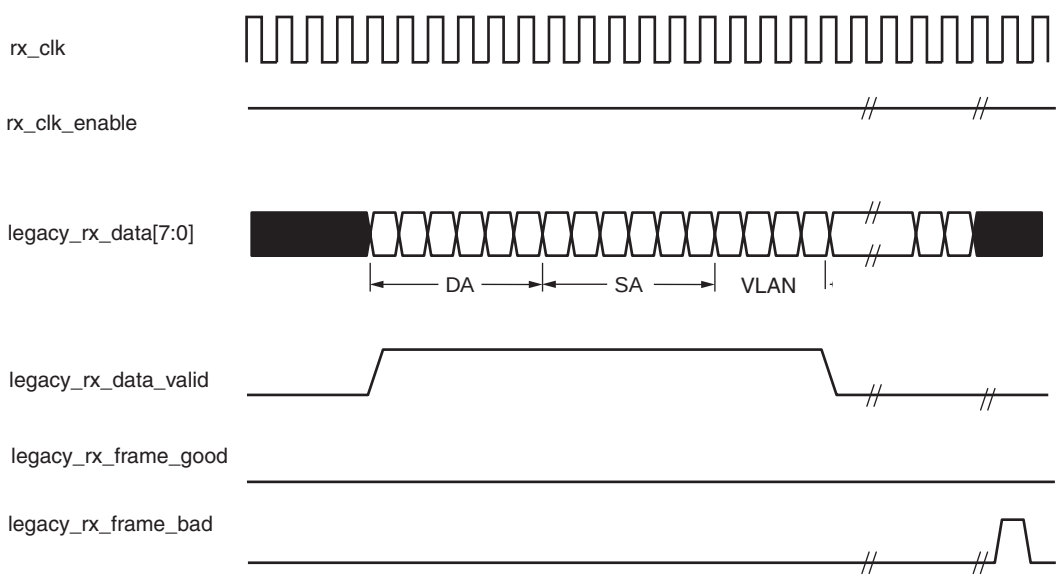

Figure 7-2: **Errored Frame Reception across the Legacy Traffic Interface**

As illustrated in Figure 7-2, reception of any frame in which the `legacy_rx_frame_bad` is asserted (in place of `legacy_rx_frame_good`) indicates that this frame must be discarded by the Legacy client; it was either received with errors or was not intended for the legacy traffic interface.

Legacy MAC Header Filters

Overview of Operation

MAC Header Filters are provided on the receiver legacy traffic path as illustrated in Figure 5-1. These have a greater flexibility than the standard address filter provided in the Tri-Mode Ethernet MAC (which must be disabled). The MAC Header Filters include the ability to filter across any of the initial 16-bytes of an Ethernet frame, including the ability to filter only on the Destination Address, Length/Type Field, VLAN tag (if present), or any bit-wise match combination of the preceding. Eight individual MAC Header Filters are provided, numbered from 0 through to 7, each of which is separately configured.

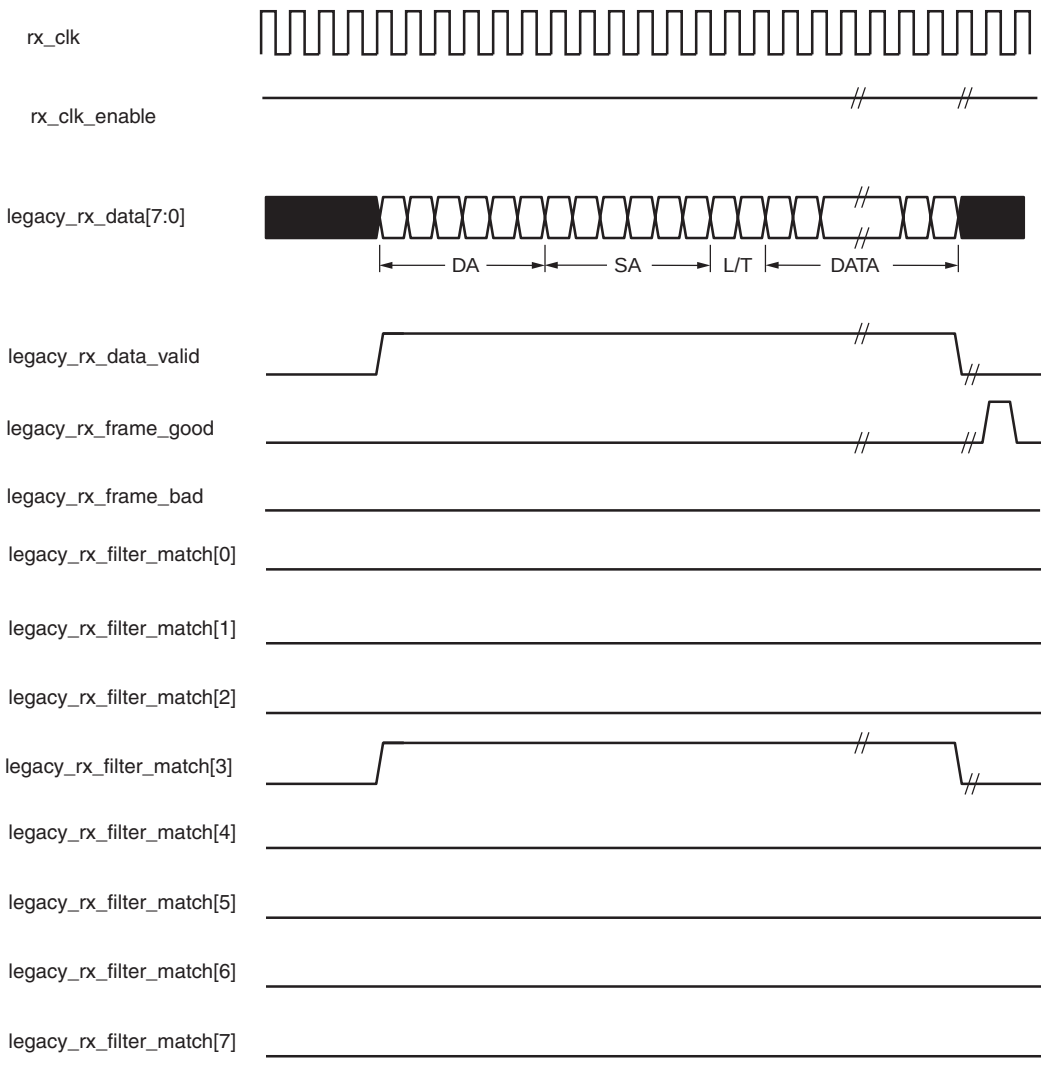

Figure 7-3: **Normal Frame Reception: Address Filter Match**

Figure 7-3 illustrates Legacy frame reception for an error free frame in which at least one of the eight individual MAC Header Filters obtained a match (filter number 3 is illustrated as having obtained the match in this example). Note the following:

- Each of the eight individual MAC Header Filters has a corresponding bit within the `legacy_rx_filter_match[7:0]` bus. If the corresponding MAC Header Filter obtains a match, the relevant bit will be asserted. This will be fully aligned with the `legacy_rx_data_valid` signal during frame reception.
- Every bit within the `legacy_rx_filter_match[7:0]` bus will be asserted for frame reception in which the Frame Destination Address (DA) contained a Broadcast Address.
- Every bit within the `legacy_rx_filter_match[7:0]` bus will be asserted when the MAC Header Filter is operating in *Promiscuous Mode* (see "Rx Filtering Control Register").

MAC Header Filter Configuration

The MAC Header Filters can be enabled or disabled by using the "Rx Filtering Control Register." This contains a *Promiscuous Mode* bit, which:

- when **enabled** allows all frames to be received on the Legacy Rx Traffic I/F.
- when **disabled** only allows frames to be received on the Legacy Rx Traffic I/F that contain a MAC Header that has matched at least one of the eight individual MAC Header Filters.

Each of the eight MAC Header Filters can be separately configured (see "MAC Header Filter Configuration"). As defined in this section, each of the eight MAC Header Filters contains two 128-bit wide registers (16-bytes):

- **Match Pattern Register**. This pattern is compared to the initial 128-bits received in the Legacy Ethernet frame (bit 0 is the first bit within the frame to be received).
- **Match Enable Register**. Each bit within this register refers to the same bit number within the Match Pattern Register. When a bit in the Match Enable Register is set to:
  - **logic 1**, the same bit number within the Match Pattern Register *is* compared with the respective bit in the received frame and must match if the overall MAC Header Filter is to obtain a match.
  - **logic 0**, the same bit number within the Match Pattern Register is *not* compared. This effectively turns the respective bit in the Match Pattern Register into a don't care bit: the overall MAC Header Filter is capable of obtaining an overall match even if this bit did not compare.

The overall result of the Match Pattern Register and Match Enable Register is to provide a highly configurable and flexible MAC Header matching logic as the "Single MAC Header Filter Usage Examples" demonstrates.

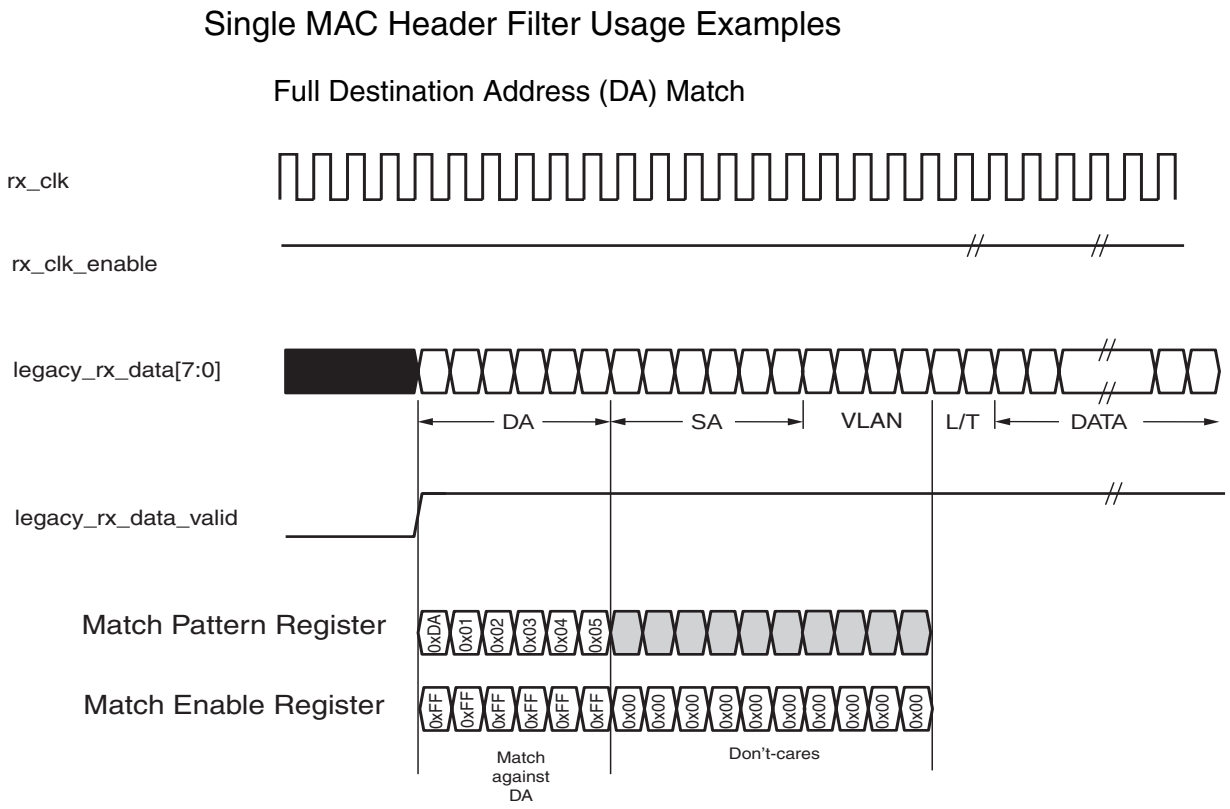

Figure 7-4: **Filtering of Frames with a Full DA Match**

The example illustrated in Figure 7-4 shows a single MAC Header Filter (one of the eight provided) configured to filter on a Destination Address. In order for the frame to obtain a match, the initial 48-bits of the received frame must exactly match the first 48-bits of the Match Pattern Register.

This example provides backwards compatibility with the Address Filters provided in the Tri-Mode Ethernet MAC (which must be disabled).

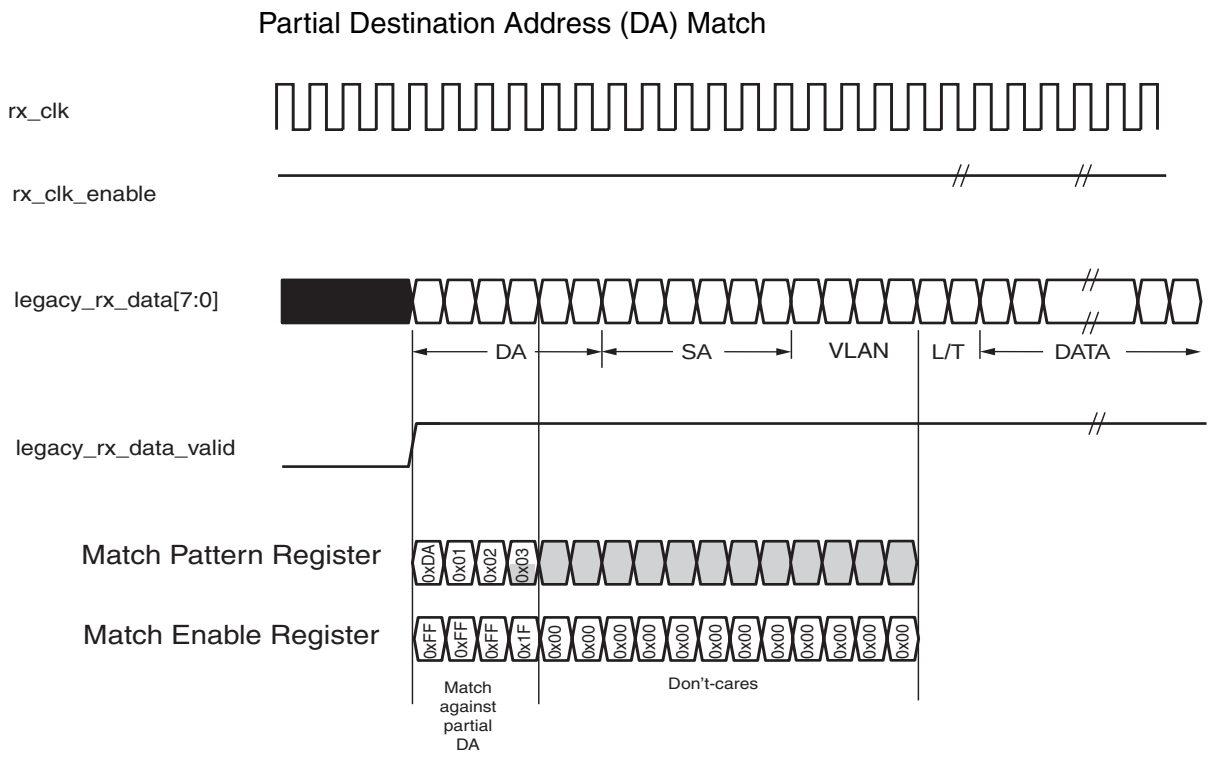

Figure 7-5: **Filtering of Frames with a Partial DA Match**

The example illustrated in Figure 7-5 shows a single MAC Header Filter (one of the eight provided) configured to filter on a partial Destination Address. In order for the frame to obtain a match, the initial 29-bits (as used in this example) of the received frame must exactly match the first 29-bits of the Match Pattern Register.

This functionality is useful for filtering across Multicast group Addresses.

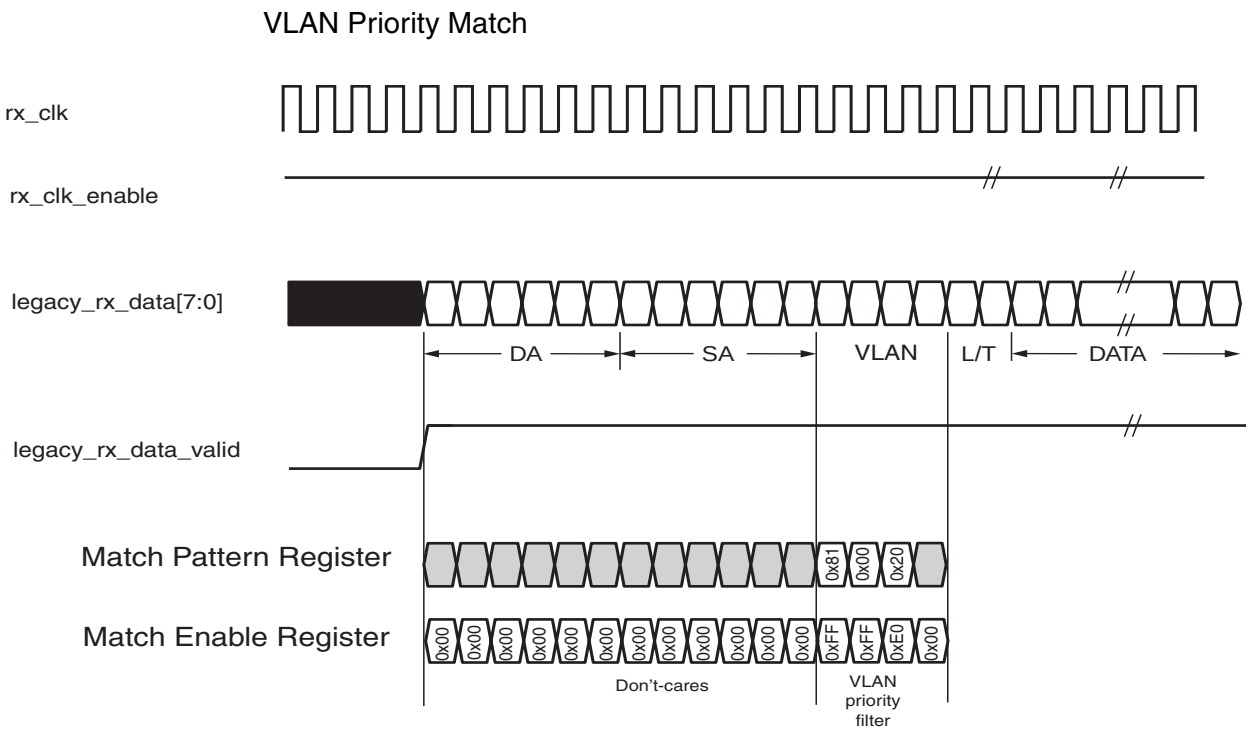

Figure 7-6: **Filtering of VLAN Frames with a Specific Priority Value**

The example illustrated in Figure 7-6 shows a single MAC Header Filter (one of the eight provided) configured to filter on frames containing a VLAN tag with a VLAN Priority value of 1.

Any Other Combinations

Because the Match Pattern Register and Match Enable Register provide the ability to filter across any bitwise match/don't-care pattern of the initial 128-bits of an Ethernet frame, match combinations of Destination Address, Length/Type Field (when no VLAN tag is present), VLAN fields (when present) can be selected with complete flexibility.

Rx AV Traffic I/F

The signals forming the Rx AV Traffic I/F are defined in Table 5-5. all signals are synchronous to the Tri-Mode Ethernet MAC receiver clock, `rx_clk`, which must always be qualified by the corresponding clock enable, `rx_clk_en` (see Table 5-1).

This interface is intentionally *identical* to the legacy receiver interface (there is a one-to-one correspondence between signal names when the `legacy_` prefix is exchanged for the `av_` prefix).

Error Free AV Traffic Reception

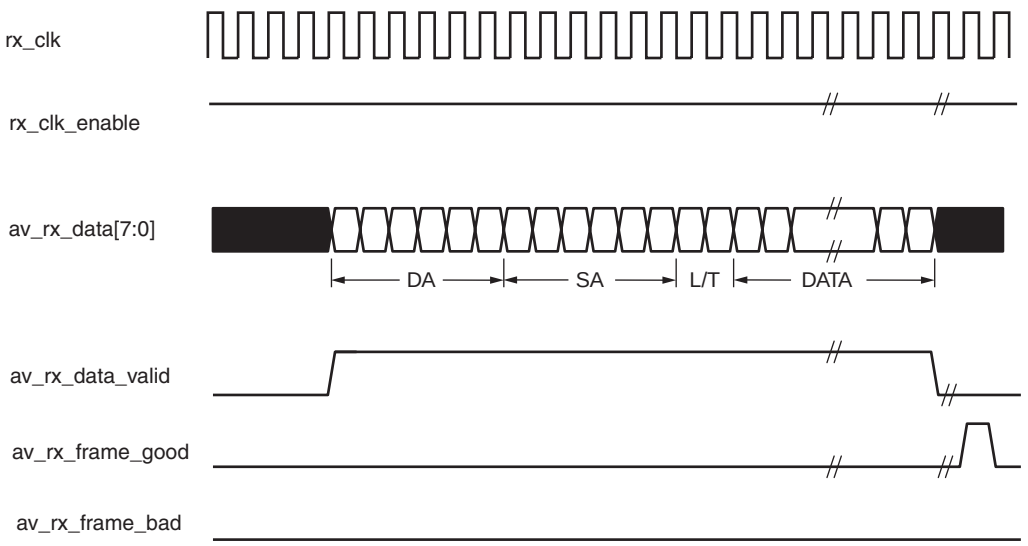

Figure 7-7: **Normal Frame Reception across the AV Traffic Interface**

Figure 7-7 illustrates the timing of a normal inbound frame transfer. The AV client must be prepared to accept data at any time; there is no buffering within the core to allow for latency in the receive client. After frame reception begins, data is transferred on consecutive clock enabled cycles to the AV receive client until the frame is complete. The core asserts the `av_rx_frame_good` to indicate that the frame was intended for the AV traffic client, and was successfully received without error.

Errored AV Traffic Reception

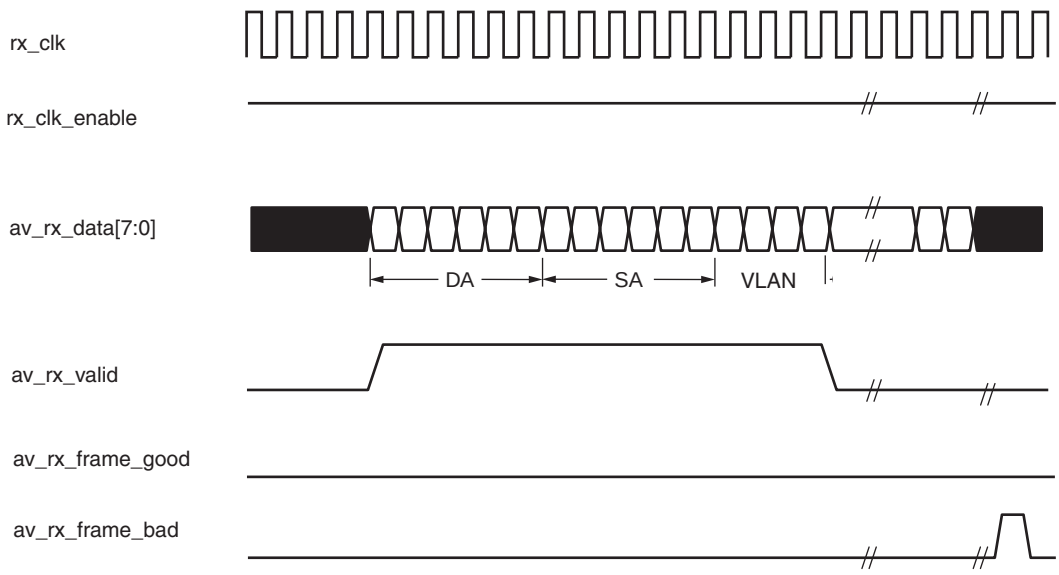

Figure 7-8: **Errored Frame Reception across the AV Traffic Interface**

As illustrated in Figure 7-8, reception of any frame in which the `av_rx_frame_bad` is asserted (in place of `av_rx_frame_good`) indicates that this frame must be discarded by the AV client; it was either received with errors or was not intended for the AV traffic interface.

Chapter 8

Real Time Clock and Time Stamping

This chapter considers two of the logical components that are partially responsible for the AVB timing synchronization protocol.

- "Real Time Clock"
- "Time Stamping Logic"

These are both described in this chapter as they are closely related.

Real Time Clock

A significant component of the PTP network wide timing synchronization mechanism is the Real Time Counter (RTC), which provides the common time of the network. Every device on the network will maintain its own local version.

The RTC is effectively a large counter which consists of a 32-bit nanoseconds field (the unit of this field is 1 nanosecond and this field will count the duration of exactly one second, then reset back to zero) and a 48-bit seconds field (the unit of this field is one second: this field will increment when the nanosecond field saturates at 1 second). The seconds field only wraps around when its count fully saturates. The entire RTC is therefore designed never to wrap around in our lifetime. The RTC is summarized in Figure 8-1.

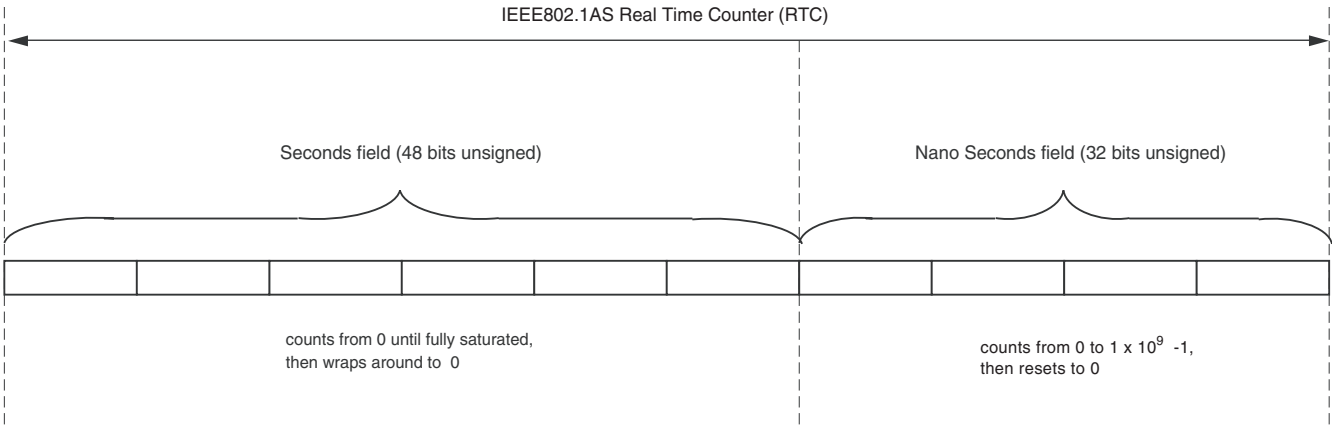

Figure 8-1: **Real Time Counter (RTC)**

Conceptually, the RTC is not related to the frequency of the clock used to increment it. A configuration register within the core provides a configurable increment rate for this counter: this increment register,"RTC Increment Value Control Register," is for this reason simply programmed with the value of the RTC Reference clock period which is being used to increment the RTC. The resolution of this increment register is very fine (in units of 1/1048576 ($1/2^{20}$) fraction of one nanosecond). Therefore, the RTC increment rate can be adjusted to a very fine degree of accuracy. This provides the following features:

- The RTC can be incremented from any available clock frequency that is greater than the AVB standards defined minimum of 25 MHz. However, the faster the frequency of the clock, the smaller will be the step increment and the smoother will be the overall RTC increment rate. Xilinx recommends clocking the RTC logic at 125 MHz because this is a readily available clock source (obtained from the transmit clock source of the Ethernet MAC at 1 Gbps speed): this frequency will significantly exceed the minimum performance of the P802.1AS specification.
- When acting as a clock slave, the rate adjustment of the RTC can be matched to that of the network clock master to an exceptional level of accuracy (by slightly increasing or decreasing the value within the "RTC Increment Value Control Register"). The software drivers provided with this core will periodically calculate the increment rate error between itself and the master, and update the RTC increment value accordingly.

The core also contains configuration registers, "RTC Offset Control Registers," which allow a large step change to be made to the RTC. This can be used to initialize the RTC, after power-up. It is also used to make periodic corrections, as required, by the software drivers when operating as a clock slave: however, if the increment rates are closely matched, these periodic step corrections will be small.

RTC Implementation

Increment of Nanoseconds Field

Figure 8-2 illustrates the implementation used to create the RTC nanoseconds field. This is performed by the use of an implementation specific 20-bit *sub-nanoseconds field* as illustrated. The nanoseconds and sub-nanoseconds fields can be considered to be concatenated together.

All RTC logic within the core is synchronous to the RTC Reference Clock, `rtc_clk`.

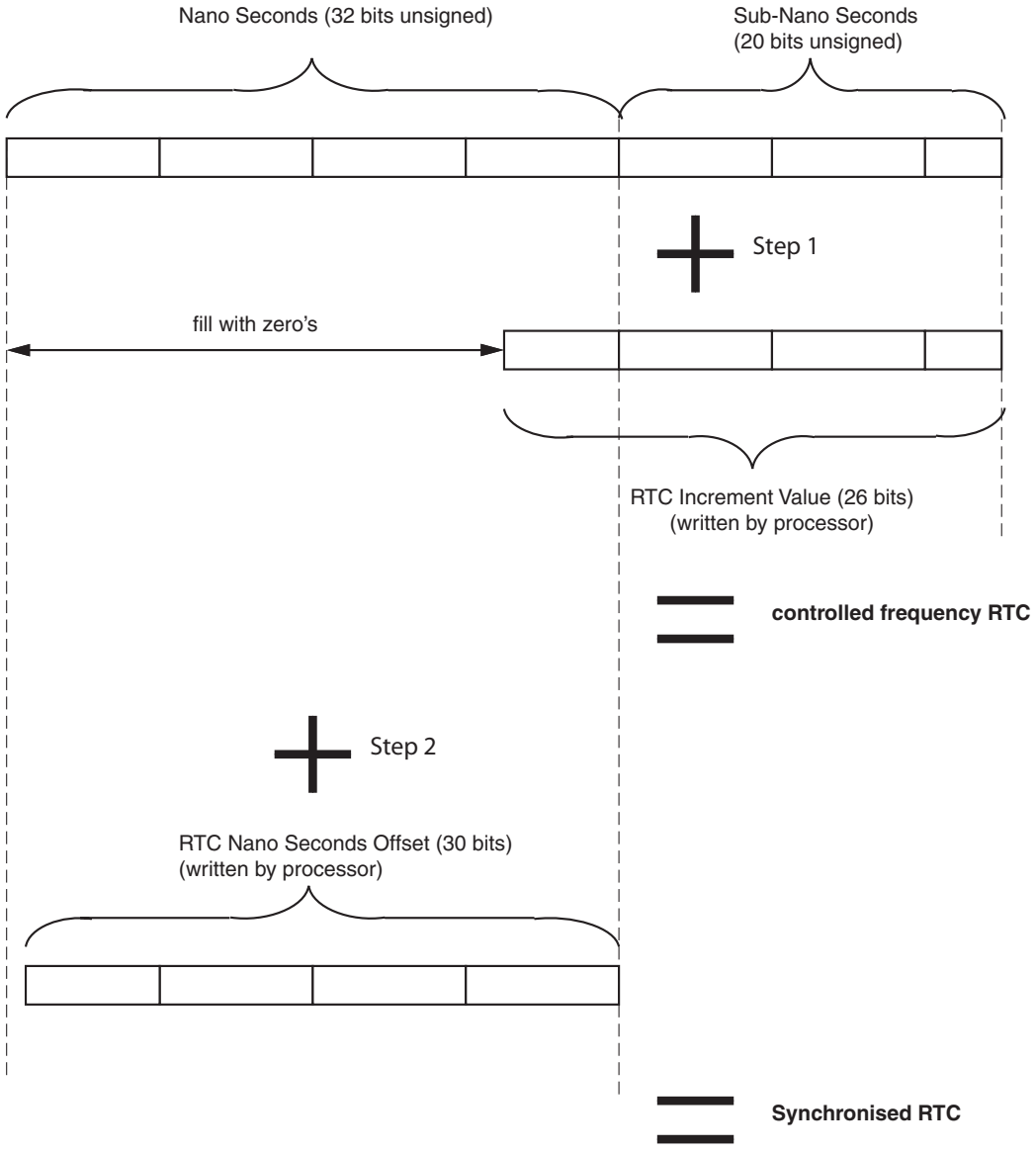

Figure 8-2: **Increment of Sub-nanoseconds and Nanoseconds Field**

There are two stages to the implementation:

(Step 1) Controlled Frequency RTC

The RTC Increment Value illustrated in Figure 8-2 is set directly from the "RTC Increment Value Control Register." The upper 6 bits of this register align with the lower 6 bits of the RTC nanoseconds field. The lower 20-bits of the RTC Increment Value align with the 20-bit sub-nanoseconds field. It is assumed that the frequency of the RTC reference clock is known by the processor to enable the increment value to be programmed correctly. For example, if the RTC is being clocked from a 125 MHz clock source, a nominal increment value of 8 ns should be programmed (by writing the value 0x800000 into the "RTC Increment Value Control Register"). However, if the microprocessor determines that this clock is drifting with respect to the grand master clock, it can revise this nominal 8 ns up or down by a very fine degree of accuracy.

The "step 1" addition illustrated in Figure 8-2 (of current counter value plus increment) will occur on every clock cycle of the RTC reference clock. The result from this addition forms the new value of the "controlled frequency RTC" nanoseconds field. This controlled frequency RTC will initialize to zero, following reset, and will continue to increment smoothly on every RTC reference clock cycle by the current value contained in the RTC Increment Value Control Register.

Figure 8-2 illustrates that 26 bits have been reserved for the Increment Value, the upper 6-bits of which overlap into the nanoseconds field. For this reason, the largest per-cycle increment = 1ns * 2^6 = 64 ns. The lowest clock period which is expected to increment this counter is 40 ns (corresponding to the 25 MHz MAC clock used at 100 Mbps speeds). So this should satisfy all allowable clock periods.

(Step 2) Synchronized RTC

The value contained in the "RTC Offset Control Registers" written by the microprocessor, is then applied to the free running "controlled frequency RTC" counter. This is used by the microprocessor to:

- Initialize the power-up value of the Synchronized RTC.
- Apply step corrections to the Synchronized RTC (when a slave), based on the timing PTP packets received from the Grand Master Clock RTC.

The "step 2" addition illustrated in Figure 8-2 (of controlled frequency RTC value plus offset) will occur on every clock cycle of the RTC reference clock. The result from this addition forms the new value of the Synchronized RTC nanoseconds field. It is this version of the RTC nanoseconds field which is made available as an output of the core - the `rtc_nanosec_field[31:0]` port.

Increment of the Seconds Field

The RTC seconds field is, conceptually, implemented in a similar way to the nanoseconds field. The seconds field should be incremented by a value of one whenever the synchronized RTC nanoseconds field saturates at one-second. The "RTC Offset Control Registers" allow the software to make large step corrections to the seconds field in a similar manner. Again, the step correction capability can be used to either initialize the RTC counter following reset, or to synchronize the local RTC to that of the Grand Master Clock (when the local device is acting as a clock slave).

Clock Outputs Based on the Synchronized RTC Nanoseconds Field

The `clk8k` (8 kHz clock) output, derived from the Synchronized RTC, is provided as an output from the core. The synchronized RTC counter, unlike the controlled frequency version, has no long-term drift (assuming the provided software drivers are used correctly). Therefore, the `clk8k` signal will be synchronized exactly to the network RTC frequency.

The 8 kHz clock is the period of the shortest class measurement interval for an SR class as specified in IEEE802.1Qav. This clock could also be useful for external applications (for example, a 1722 implementation of the AV traffic).

Time Stamping Logic

Whenever a PTP packet, used with the Precise Timing Protocol (PTP), is transmitted or received (see "Precise Timing Protocol Packet Buffers" in Chapter 9), a sample of the current value of the RTC is taken and made available for the software drivers to read. The hardware makes no distinction between frames carrying event or general PTP messages (as defined in IEEE P802.1AS); it will always store a timestamp value for ethernet frames containing the Ethertype specified for PTP messages.

This time stamping of packets is a key element of the tight timing synchronization across the AVB network wide RTC, and these samples must be performed in hardware for accuracy. The hardware in this core will therefore sample and capture the local nanoseconds RTC field for every PTP frame transmitted or received. These captured time stamps are stored in the "Precise Timing Protocol Packet Buffers" alongside the relevant PTP frame, and are read and used by the PTP software drivers.

It is important to realize that is it actually the "controlled frequency RTC" nanoseconds field which is sampled by the time stamping logic rather than the synchronized RTC (see Figure 8-2). This is important when operating as a clock slave: the controlled frequency RTC always acts as a smooth counter whereas the synchronized RTC may suffer from occasional step changes (whenever a new offset adjustment is periodically applied by the software drivers). These step changes, avoided by using the controlled frequency RTC, could otherwise lead to errors in the various PTP calculations which are performed by the software drivers.

Note: The "Software Drivers" can themselves obtain (when required) the local synchronized RTC value simply by summing the captured time stamp with the current nanoseconds offset value of the "RTC Offset Control Registers" (effectively performing the step 2 calculation of Figure 8-2 in software).

Time Stamp Sampling Position of MAC Frames

A time stamp value should be sampled at the beginning of the first symbol following the Start of Frame Delimiter (SFD) of the Ethernet MAC frame as seen on the PHY. This is illustrated in Figure 8-3.

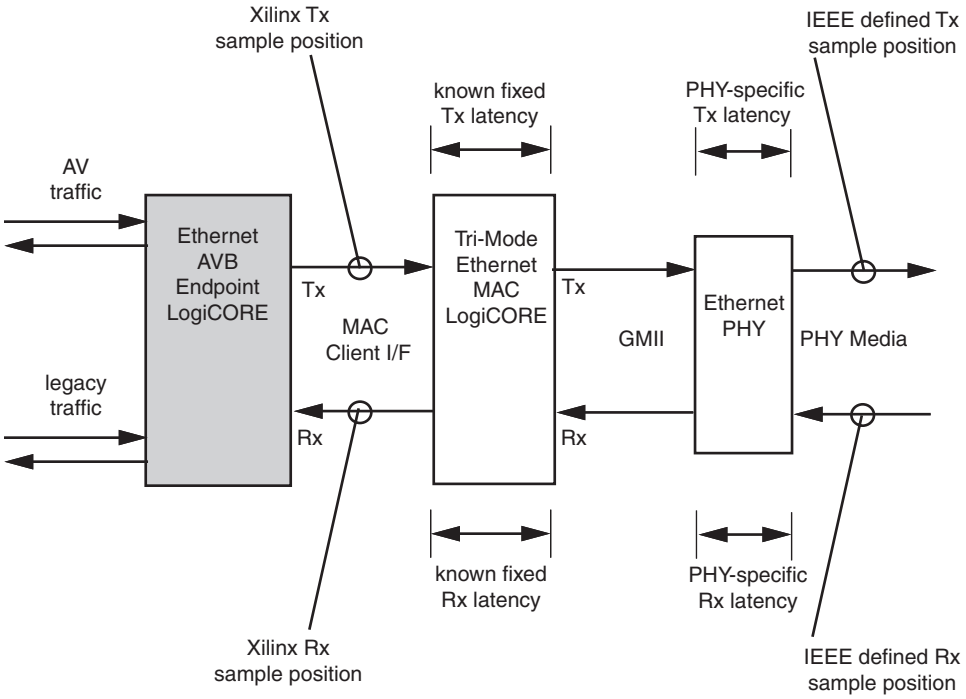

Figure 8-3: **Time Stamping Position**

Figure 8-3 also illustrates the actual time stamp sampling position that is used by the core. Time stamps are taken after the MAC frame SFD is seen not on the GMII, but on the MAC Client I/F. The time stamping logic is deliberately designed this way for the following reasons:

1. When the Ethernet AVB Endpoint core is to be connected to the Embedded Tri-Mode Ethernet MAC, the GMII is not always available to the FPGA fabric logic: specifically when used with a 1000BASE-X or SGMII physical interface, the GMII exists only as an internal connection within the embedded block. Therefore, by sampling on the client interface, we enable the Ethernet AVB Endpoint core to be connected to ANY Xilinx Tri-Mode MAC used in ANY configuration.
2. Sampling on the MAC Client I/F provides the Ethernet AVB Endpoint core with the required time stamp exactly when it is needed. Sampling on the GMII would require the use of sideband Time stamp Value FIFOs (there may be more than a single MAC frame present in the pipeline stages of the MAC transmitter or receiver). So by sampling on the MAC Client I/F, we are also able to reduce the need for extra FIFO logic.

Because the Xilinx Tri-Mode Ethernet MACs have a known fixed latency, the time stamps taken can easily be translated into the equivalent GMII position to comply with the standard. This is performed in the software drivers where the MAC transmitter and receiver latencies are held in #defines in a header file. The software drivers also contain placeholder #defines for users to input the PHY-specific latency values for the PHYs used in the system.

IEEE1722 Real Time Clock Format

The IEEE1722 specification defines the *avbtp_timestamp* field. This is derived by sampling the IEEE802.1 AS Real Time Clock and converting the low order time to nanoseconds. From version 2.1 onwards, this conversion is now performed in the Ethernet AVB Endpoint core and an alternative RTC, in the 1722 format, is output on the `rtc_nanosec_field_1722[31:0]` port.

This port contains a 32-bit word representing nanosecond values. Unlike the IEEE802.1 AS nanosecond field (which resets back to zero when it reaches 1 second), the IEEE1722 nanosecond field counts fully to 0xFFFFFFFF before wrapping around. The field therefore wraps around approximately every 4 seconds.

If the system is using the IEEE1722 functionality, this port can be sampled to create the *avbtp_timestamp* field. Otherwise this port can be ignored.

XILINX®

Chapter 9

Precise Timing Protocol Packet Buffers

This chapter considers two of the logical components which are partly responsible for the AVB timing synchronization protocol.

- "Tx PTP Packet Buffer"
- "Rx PTP Packet Buffer"

These are both described in this chapter as they are closely related.

Tx PTP Packet Buffer

The Tx PTP packet buffer is illustrated in Figure 9-1. This packet buffer provides working memory to hold the PTP frames which are required for transmission. The software drivers, via the PLB configuration bus, can read/modify/write the PTP frame contents, and whenever required, can request transmission of the appropriate PTP frames.

The PTP packet buffer is implemented in dual-port block RAM. Port A of the block RAM is connected to the PLB configuration bus: all addresses in the buffer are read/writable through the PLB. Port B of the block RAM is connected to the Tx Arbiter module, allowing PTP frames to be read out of the block RAM and transmitted through the connected TEMAC.

The Tx PTP Packet Buffer is divided into eight identical buffer sections as illustrated. Each section contains 256 bytes, which are formatted as follows:

- the first byte, at address zero, contains a frame length field. This indicates how many bytes make up the PTP frame that is to be transmitted from this particular PTP buffer.
- The next seven bytes, from address 1 to 7, are reserved for future use.
- The PTP frame data itself is stored from address 8 onwards. The amount of addresses used is dependent on the indicated frame length field, which will be different for each PTP frame type. Each PTP buffer provides a maximum of 244 bytes (more than that required for the largest PTP frame). Each PTP frame holds the entire MAC frame (with the exception of any required MAC padding or CRC - these will automatically be inserted by the TEMAC) from the Destination Address field onwards.
- The top four addresses of each buffer, from address 0xFC to 0xFF are reserved for a time stamp field. At the beginning of PTP frame transmission from any of the eight buffers, the "Time Stamping Logic" will sample the "Real Time Clock". Following the end of PTP frame transmission, this captured timestamp will automatically be written into this location to accompany the frame for which it was taken.

Despite the logic and formatting of each individual PTP buffer being identical, the block RAM is pre-initialized at device configuration to hold template copies of each of the PTP frames, as indicated in Figure 9-1. This shows that the first seven memory segments are in use. PTP Buffer number 8 is currently unused and could therefore be used by proprietary applications.

The "Tx PTP Packet Control Register" is defined for the purpose of requesting which of the eight Tx PTP Buffers are to be transmitted. It is possible to request more than a single frame at one time (indeed it is possible to request all 8). When more than one frame is requested, the Tx PTP Buffer logic will give a priority order to the lowest PTP Buffer Number that has been requested.

The "Tx PTP Packet Control Register" also contains a frame waiting field. This can be read by the software drivers to determine which of the previously requested PTP frames have been sent, and which are still queued.

Following transmission completion of each requested PTP frame, a dedicated interrupt signal, `interrupt_ptp_tx`, will be generated by the core. On the assertion of the interrupt, the captured timestamp will already be available in the upper four bytes of the buffer, and the tx_packet field of the"Tx PTP Packet Control Register" will indicate the most recently transmitted Buffer Number.

The "Software Drivers" provided with the core, using the PLB and dedicated interrupts, will use this interface to periodically, as defined by the IEEE802.1AS protocol, update specific fields within the PTP packets, and request transmission of these packets.

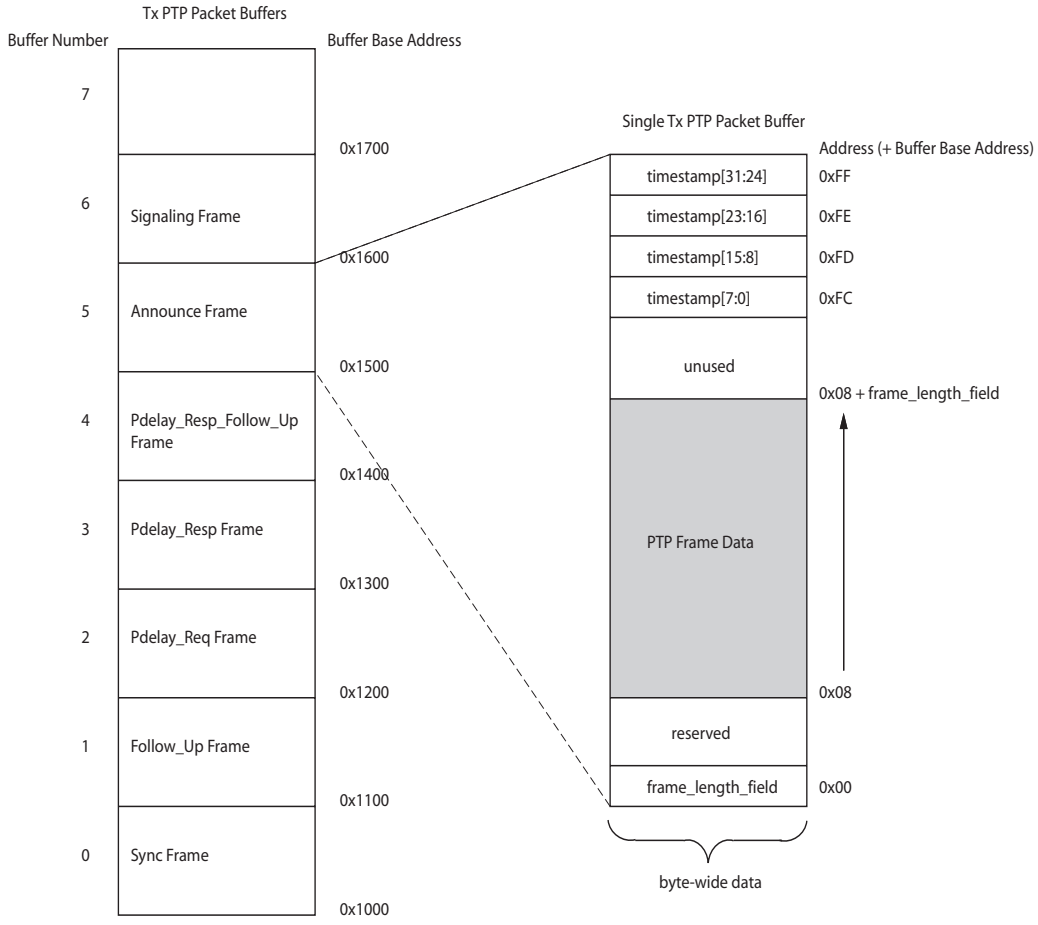

Figure 9-1: **Tx PTP Packet Buffer Structure**

Rx PTP Packet Buffer

The Rx PTP packet buffer is illustrated in Figure 9-2. This provides working memory to hold each received PTP frame. The software drivers, via the PLB configuration bus, can then read and decode the contents of the received PTP frames.

The PTP packet buffer is implemented in dual-port block RAM. Port A of the block RAM is connected to the PLB configuration bus: all addresses in the buffer can be read (writes are not allowed). Port B of the block RAM is connected to the Rx Splitter module, which routes all received PTP frames into the Rx PTP Packet Buffer.

The Rx PTP Packet Buffer is divided into sixteen identical buffer sections as illustrated. Each section contains 256 bytes, which are formatted as follows:

- The PTP frame data itself is stored from address 0 onwards: the entire MAC frame from the Destination Address onwards will be written (with the exception of the FCS field which will have been removed by the TEMAC). The amount of addresses used will be dependent on the particular PTP frame size, which is different for each PTP frame type. Each PTP buffer provides a maximum of 252 bytes (more than that required for the largest PTP frame). Should an illegally oversized PTP frame be received, the first 252 bytes will be captured and stored - other bytes will be lost.
- The top four addresses of each buffer, from address 0xFC to 0xFF are reserved for a timestamp field. At the beginning of PTP frame reception, the "Time Stamping Logic" will sample the "Real Time Clock." Following the end of PTP frame reception, this captured timestamp will automatically be written into this location to accompany the frame for which it was taken.

Following reset, the first received PTP frame will be written into Buffer Number 0. The next subsequent received PTP frame will be written into the next available buffer - in this case number 1. This process continues with buffer number 2, 3, then 4, and so forth, being used. After receiving the 16th PTP frame (which would have been stored into buffer number 15), the count will be reset, and then buffer number 0 will be overwritten with the next received PTP frame. For this reason, at any one time, the Rx PTP Packet Buffer is capable of storing the most recently received sixteen PTP frames.

Following the completion of PTP frame reception, a dedicated interrupt signal, `interrupt_ptp_rx`, will be generated by the core. On the assertion of the interrupt, the captured timestamp will already be available in the upper four bytes of the buffer, and the rx_packet field of the "Rx PTP Packet Control Register" will indicate the most recently filled Buffer Number.

The "Software Drivers" provided with the core, using the PLB and dedicated interrupt, will use this interface to decode, and then act on, the received PTP packet information.

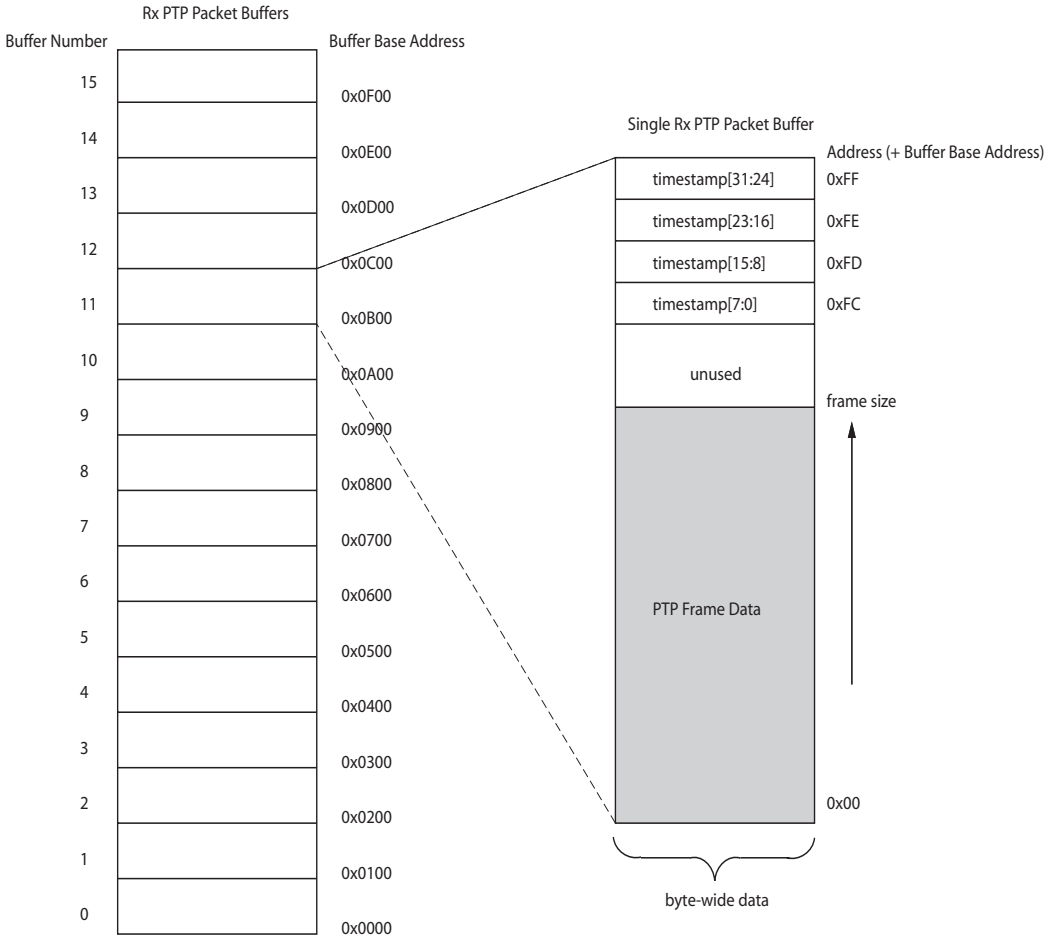

Figure 9-2: **Rx PTP Packet Buffer**

XILINX®

Chapter 10

Configuration and Status

This chapter provides general guidelines for configuring and monitoring the Ethernet AVB Endpoint core, including an introduction to the PLB configuration bus and a description of the core management registers.

Processor Local Bus Interface

The Processor Local Bus (PLB) bus on the Ethernet AVB Endpoint core is designed to be integrated directly in the Xilinx Embedded Development Kit (EDK) where it can be easily integrated and connected to the supported embedded processors (MicroBlaze™ or PowerPC®). As a result, the PLB interface does not require in-depth understanding and the following information is provided for reference only. See the EDK documentation for further information.

The PLB interface, defined by IBM, can be complex and support many usage modes (such as multiple bus masters). It can support single or burst read/writes, and can support different bus widths and different peripheral bus widths.

The general philosophy of the Ethernet AVB Endpoint core has been to implement a PLB interface which is as simple as possible. The following features are provided:

- 32-bit data width.
- Implements a simple PLB slave.
- Supports single read/writes only (no burst or page modes).

Single Read Transaction

Figure 10-1 illustrates a single read data transfer on the PLB. Note the following:

- Wait states can be added to the Address cycle by asserting `Sl_wait` and delaying `Sl_addrAck`.
- Wait states can be inserted in the Read fetch by delaying the assertion of `Sl_rdDAck`.

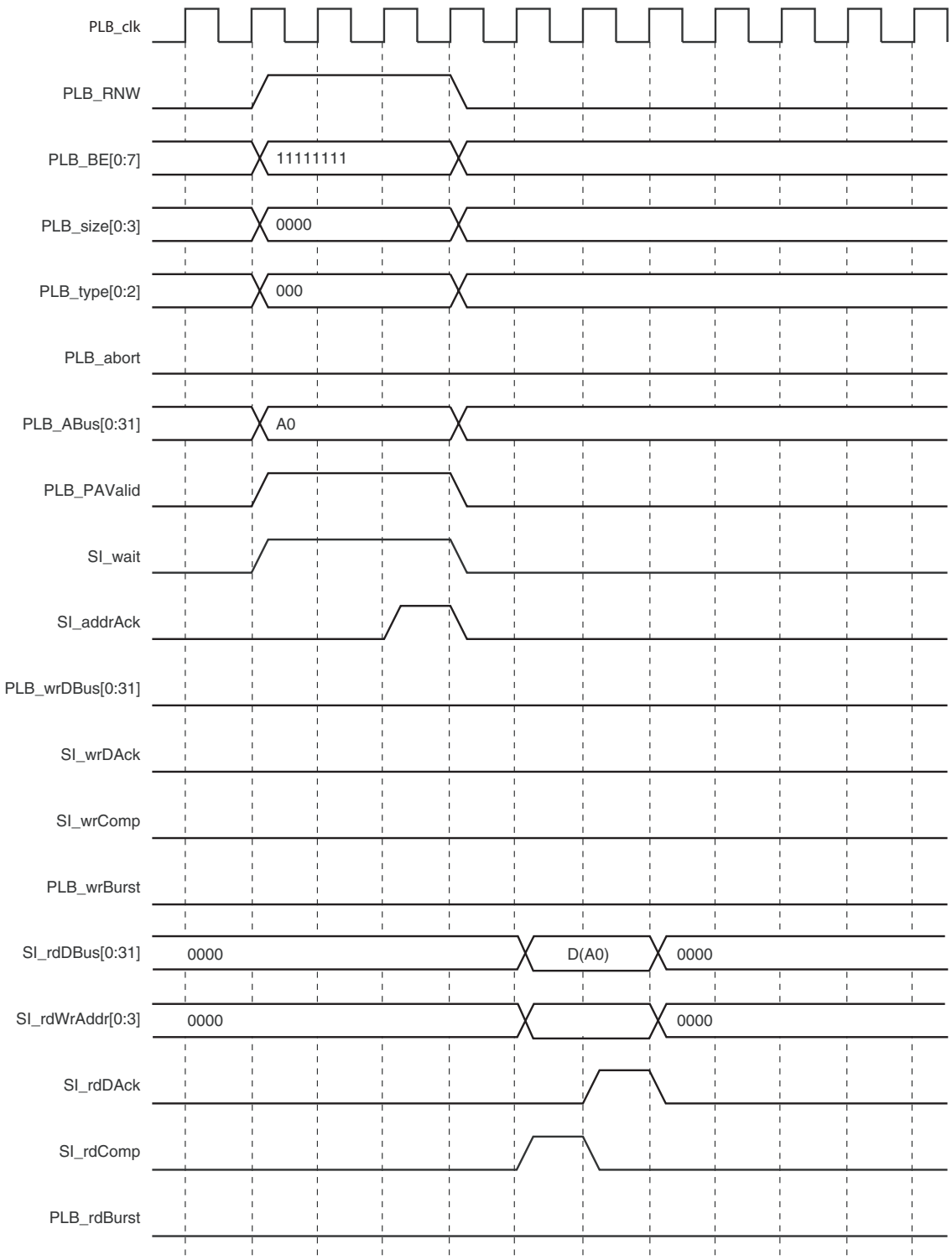

Figure 10-1: **Single Read Transaction**

Single Write Transaction

Figure 10-2 illustrates a single write data transfer on the PLB. Note the following:

- Wait states can be added to the Address cycle by asserting `Sl_wait` and delaying `Sl_addrAck`.
- Wait states can be inserted in the Write sample by delaying the assertion of `Sl_wrDAck`.

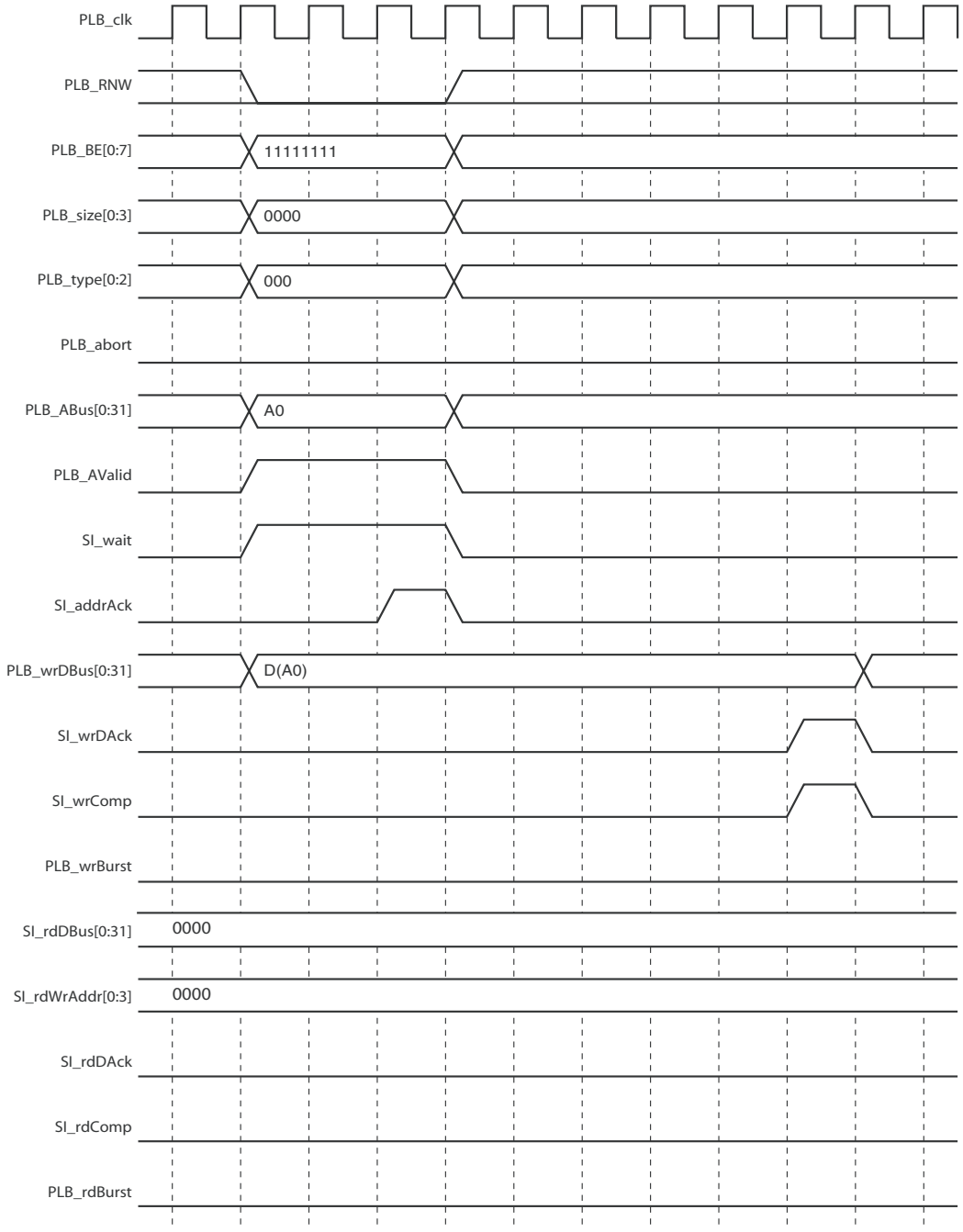

Figure 10-2: **Single Write Transaction**

PLB Address Map and Register Definitions

Figure 10-3 displays an overview of the Address Space occupied by the Ethernet AVB Endpoint core on the PLB. Common across all addressable space, each unique PLB address value references a single *byte* of data.

The variable `PLB_base_address` shown in Figure 10-3 and in the tables that follow represent the starting (base) address of the AVB core within the entire PLB address space; this is:

- selected from the CORE Generator™ software Customization GUI (see "PLB Base Address" in Chapter 4) when the core is generated in "Standard CORE Generator Format".
- automatically assigned and configured when the core is generated in "EDK pcore Format".

The entire address space is now described in two sections:

- "Ethernet AVB Endpoint Address Space"
- "Tri-Mode Ethernet MAC Address Space" (which can be addressed through the Ethernet AVB Endpoint core Address Space). This address is only present when the core is generated in "Standard CORE Generator Format".

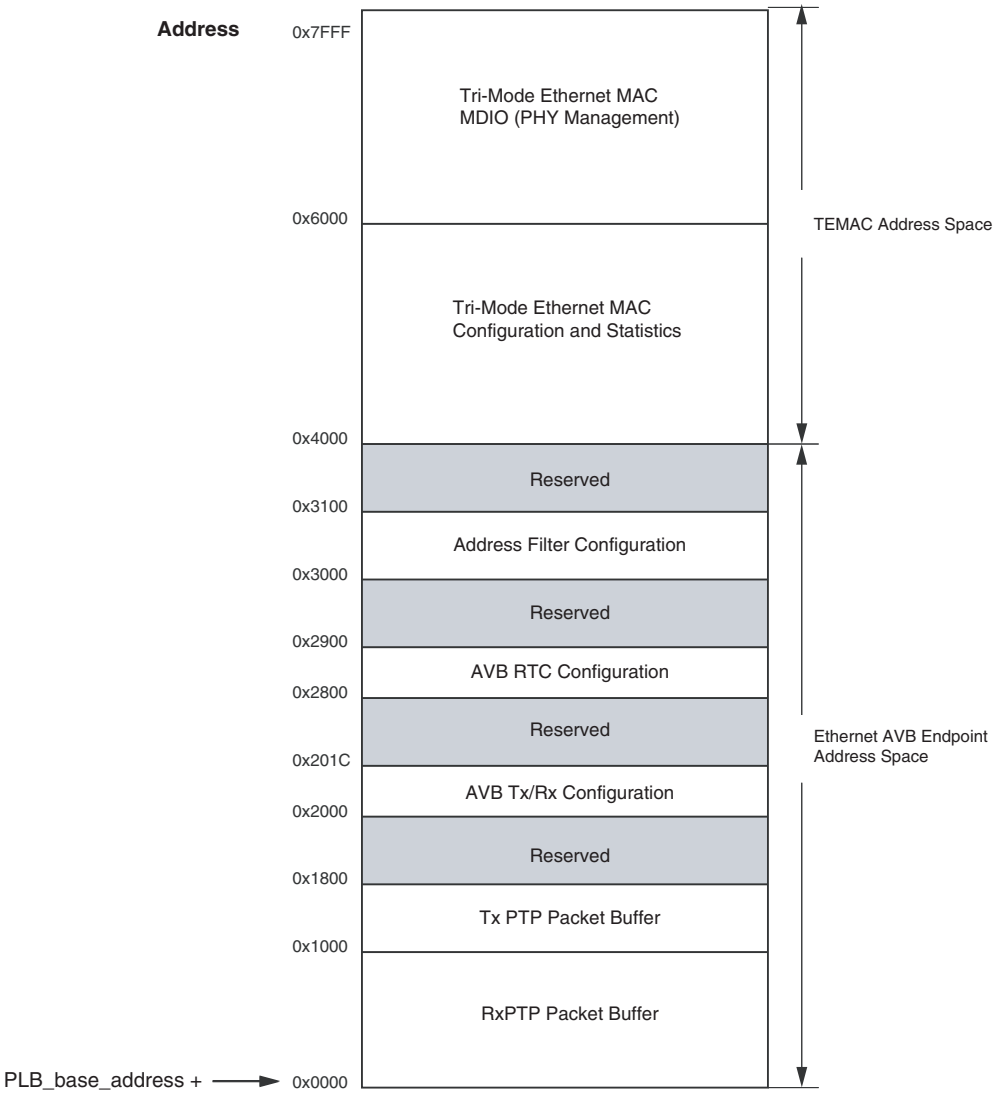

Figure 10-3: **PLB Address Space of the Ethernet AVB Endpoint Core and Connected Tri-Mode Ethernet MAC**

Ethernet AVB Endpoint Address Space

Rx PTP Packet Buffer Address Space

The Address space of the "Rx PTP Packet Buffer" is 4k bytes, from PLB_base_address to (PLB_base_address + 0x0FFF). This represents the size of a single Virtex®-5 FPGA block RAM pair (4k bytes). Every byte of this Block RAM can be read from the PLB. See "Rx PTP Packet Buffer" for operation.

Tx PTP Packet Buffer Address Space

The Address space of the "Tx PTP Packet Buffer" is continuous from (PLB_base_address + 0x1000) to (PLB_base_address + 0x17FF), representing the size of a single Virtex-5 FPGA Block 18k RAM (2k bytes). Every byte of this Block RAM is read/write accessible via the PLB. See "Tx PTP Packet Buffer" for operation.

Ethernet Audio Video End Point Configuration Registers

Tx PTP Packet Control Register

Table 10-1 defines the associated control register of the "Tx PTP Packet Buffer," used by the "Software Drivers" to request the transmission of the PTP frames.

Table 10-1: **Tx PTP Packet Buffer Control Register (PLB_base_address + 0x2000)**

Bit no	Default	Access	Description
7-0	0	WO	**tx_send_frame** bits. The Tx PTP Packet Buffer is split into 8 regions of 256 bytes. Each of these can contain a separate PTP frame. There is 1 tx_send_frame bit for each of the 8 regions. Each bit, when written to '1', will cause a request to be made to the "Tx Arbiter." When access is granted, the frame contained within the respected region will be transmitted. If read, will always return 0.
15-8	0	RO	**tx_frame_waiting** indication. The Tx PTP Packet Buffer is split into 8 regions of 256 bytes, each of which can contain a separate PTP frame. There is 1 tx_frame_waiting bit for each of the 8 regions. Each bit, when logic 1, indicates that a request has been made for frame transmission to the "Tx Arbiter," but that a grant has not yet occurred. When the frame has been successfully transmitted, the bit will be set to logic 0. This bit allows the microprocessor to run off a polling implementation as opposed to the Interrupts.
18-16	0	RO	**tx_packet**. indicates the number (block RAM bin position) of the most recently transmitted PTP packet.
31-19	0	RO	Unused

Note: A read or a write to this register clears the interrupt_ptp_tx interrupt (asserted after each successful PTP packet transmission).

Rx PTP Packet Control Register

Table 10-2 defines the associated control register of the "Rx PTP Packet Buffer," used by the "Software Drivers" to monitor the position of the most recently received PTP frame.:

Table 10-2: **Rx PTP Packet Buffer Control Register (PLB_base_address + 0x2004)**

Bit no	Default	Access	Description
0	0	WO	**rx_clear**. When written with a '1,' forces the buffer to empty, in practice moving the write address to the same value as the read address. If read, always return 0.
7-1	0	RO	Unused
11-8	0	RO	**rx_packet**. Indicates the number (block RAM bin position) of the most recently received PTP packet.
31-12	0	RO	Unused

Note: A read or a write to this register clears the interrupt_ptp_rx interrupt (asserted after each successful PTP packet reception).

Rx Filtering Control Register

Table 10-3 defines the associated control register of the "Rx Splitter." The Rx path is capable of identifying the AV packets using configurable VLAN priority:

Table 10-3: **Rx Filtering Control Register (PLB_base_address + 0x2008)**

Bit no	Default	Access	Description
2-0	3	R/W	**VLAN Priority A**. If a tagged packet is received with a VLAN priority field matching either of the Priority A or B values, then the packet will be considered as an AV frame: it will be passed to the AV I/F. Otherwise it will be passed to the Legacy I/F.
7-3	0	RO	Unused
10-8	2	R/W	**VLAN Priority B**. If a tagged packet is received with a VLAN priority field matching either of the Priority A or B values, then the packet will be considered as an AV frame: it will be passed to the AV I/F. Otherwise it will be passed to the Legacy I/F.
15-11	0	RO	Unused
16	1	R/W	**Promiscuous Mode** for the "Legacy MAC Header Filters." If this bit is set to 1, the MAC Header Filter is set to operate in promiscuous mode. All frames will be passed to the "Rx Legacy Traffic I/F." If set to 0 then only matching MAC headers are passed to the "Rx Legacy Traffic I/F."

Tx Arbiter Send Slope Control Register

The sendSlope variable is defined in IEEE P802.1 Qav to be the rate of change of credit, in bits per second, when the value of credit is decreasing (during AV packet transmission). Together with the "Tx Arbiter Idle Slope Control Register," registers define the maximum limit of the bandwidth that is reserved for AV traffic; this will be enforced by the "Tx Arbiter." The default values allow the maximum bandwidth proportion of 75% for the AV traffic. See the *IEEE P802.1 Qav* specification and "Tx Arbiter" for more information.

Table 10-4: **Tx Arbiter Send Slope Control Register (PLB_base_address + 0x200C)**

Bit no	Default	Access	Description
19-0	2048	R/W	The value of "sendSlope"
31-20	0	RO	Unused

Tx Arbiter Idle Slope Control Register

The idleSlope variable is defined in IEEE802.1Qav to be the rate of change of credit, in bits per second, when the value of credit is increasing (whenever there is no AV packet transmission). Together with the "Tx Arbiter Send Slope Control Register," two registers define the maximum limit of the bandwidth that is reserved for AV traffic; this is enforced by the "Tx Arbiter." The default values allow the maximum bandwidth proportion of 75% for the AV traffic. See the *IEEE P802.1 Qav* specification and "Tx Arbiter" for more information.

Table 10-5: **Tx Arbiter Idle Slope Control Register (PLB_base_address + 0x2010)**

Bit no	Default	Access	Description
31-20	0	RO	Unused
19-0	6144	R/W	The value of "idleSlope"

RTC Offset Control Registers

Table 10-6 describes the offset control register for the nanoseconds field of the "Real Time Clock," used to force step changes into the counter. When in PTP clock master mode, this can be used to set the initial value following power-up. When in PTP clock slave mode, the "Software Drivers" will use this register to implement the periodic step corrections.

This register and the registers defined in Table 10-7 and in Table 10-8 are linked. These three offset values will be loaded into the RTC counter logic simultaneously following a write to this nanosecond offset register.

Table 10-6: **RTC Nanoseconds Field Offset (PLB_base_address + 0x2800)**

Bit no	Default	Access	Description
29-0	0	R/W	30-bit offset value for the RTC nanoseconds field. Used by the microprocessor to initialize the RTC, then afterwards to perform the regular RTC corrections (when in slave mode).
31-30	0	RO	Unused

Table 10-7 describes the offset control register for the lower 32-bits of seconds field of the "Real Time Clock," used to force step changes into the counter. When in PTP clock master mode, this can be used to set the initial value following power-up. When in PTP clock slave mode, the "Software Drivers" use this register to implement the periodic step corrections.

This register and the registers defined in Table 10-6 and in Table 10-8 are linked. These three offset values will be loaded into the RTC counter logic simultaneously following a write to the nanosecond offset register defined in Table 10-6.

Table 10-7: **Seconds Field Offset bits [31:0] (PLB_base_address + 0x2808)**

Bit no	Default	Access	Description
31-0	0	R/W	32-bit offset value for the RTC seconds field (bits 31-0). Used by the microprocessor to initialize the RTC, then afterwards to perform the regular RTC corrections (when in slave mode).

Table 10-8 describes the offset control register for the upper 16-bits of seconds field of the "Real Time Clock," used to force step changes into the counter. When in PTP clock master mode, this can be used to set the initial value following power-up. When in PTP clock slave mode, the "Software Drivers" use this register to implement the periodic step corrections.

This register and the registers defined in Table 10-6 and in Table 10-7 are linked. These three offset values will be loaded into the RTC counter logic simultaneously following a write to the nanosecond offset register defined in Table 10-6.

Table 10-8: **Seconds Field Offset bits [47:32] (PLB_base_address + 0x280C)**

Bit no	Default	Access	Description
15-0	0	R/W	16-bit offset value for the RTC seconds field (bits 47-32). Used by the microprocessor to initialize the RTC, then afterwards to perform the regular RTC corrections (when in slave mode).
31-16	0	RO	Unused

RTC Increment Value Control Register

Table 10-9 describes the RTC Increment Value Control Register. This provides configurable increment rate for the "Real Time Clock" counter: this increment register should simply take the value of the clock period which is being used to increment the RTC. However, the resolution of this increment register is very fine (in units of 1/1048576 ($1/2^{20}$) fraction of one nanosecond). Therefore, the RTC increment rate can be adjusted to a very fine degree of accuracy. This provides the following features:

- The RTC can be incremented from any available clock frequency that is greater than the P802.1AS defined minimum of 25 MHz.
- When acting as a clock slave, the rate adjustment of the RTC can be matched to that of the network clock master to an exceptional level of accuracy.:

Table 10-9: **RTC Increment Value Control Register (PLB_base_address + 0x2810)**

Bit no	Default	Access	Description
25-0	0	R/W	Per `rtc_clk` clock period Increment Value for the RTC.
31-26	0	RO	Unused

Current RTC Value Registers

Table 10-10 describes the nanoseconds field value register for the nanoseconds field of the "Real Time Clock." When read, this will return the latest value of the counter.

This register and the registers defined in Table 10-11 and in Table 10-12 are linked. When this nanoseconds value register is read, the entire RTC (including the seconds field) is sampled.

Table 10-10: **Current RTC Nanoseconds Value (PLB_base_address + 0x2814)**

Bit no	Default	Access	Description
29-0	0	RO	Current Value of the synchronized RTC nanoseconds field. **Note**: A read from this register samples the entire RTC counter (synchronized) so that the Epoch and Seconds field are held static for a subsequent read.
31-30	0	RO	Unused

Table 10-11 describes the lower 32-bits of the seconds value register for the seconds field of the "Real Time Clock." When read, this returns the latest value of the counter.

This register and the registers defined in Table 10-10 and in Table 10-12 are linked. When the nanoseconds value register is read (see Table 10-10), the entire RTC is sampled.

Table 10-11: **Current RTC Seconds Field Value bits [31:0] (PLB_base_address + 0x2818)**

Bit no	Default	Access	Description
31-0	0	RO	Sampled Value of the synchronized RTC Seconds field (bits 31-0).

Table 10-12 describes the upper 16-bits of the seconds value register for the seconds field of the "Real Time Clock." When read, this returns the latest value of the counter.

This register and the registers defined in Table 10-10 and in Table 10-11 are linked. When the nanoseconds value register is read (see Table 10-10), the entire RTC is sampled.

Table 10-12: **Current RTC Seconds Field Value bits [47:32] (PLB_base_address + 0x281C)**

Bit no	Default	Access	Description
15-0	0	RO	Sampled Value of the synchronized RTC Seconds field (bits 47-32).
32-16	0	RO	Unused

RTC Interrupt Clear Register

Table 10-13 describes the control register defined for the `interrupt_ptp_timer` signal, the periodic interrupt signal which is raised by the "Real Time Clock."

Table 10-13: **RTC Interrupt Clear Register (PLB_base_address + 0x2820)**

Bit no	Default	Access	Description
0	0	WO	Write ANY value to bit 0 of this register to clear the interrupt_ptp_timer Interrupt signal. This bit always returns 0 on read.
31-1	0	RO	Unused

Phase Adjustment Register

Table 10-14 describes the Phase Adjustment Register, which has units of nanoseconds. This value is used to correct the 8k clock generation circuit when a new nanosecond offset value is written to the RTC. It additionally could be used to apply a phase offset to the clk8k signal.

The value written into this register will be loaded into the 8k clock generation circuit at the same instant as the offset is applied to the RTC counter logic, following a write to the nanosecond offset register defined in Table 10-6.

As an example of applying a phase offset, writing the value of the decimal 62500 (half of an 8 KHz clock period) to this register would invert the `clk8k` signal with respect to a value of 0. This register can therefore provide fine grained phase alignment of these signals to a 1 ns resolution.

Table 10-14: **RTC Phase Adjustment Register (PLB_base_address + 0x2824)**

Bit no	Default	Access	Description
29-0	0	R/W	ns value relating to the phase offset for the clk8k RTC derived timing signal.
31-30	0	RO	Unused

Software Reset Register

Table 10-15 describes the Software Reset Register. This register contains unique bits which can be written to in order to request the reset of a particular section of logic from within the Ethernet AVB Endpoint core. A single bit can be written to in a single CPU transaction in order to reset just that particular function; several to all bits can be written to in a single CPU transaction in order to reset several to all of the available reset functions.

Table 10-15: **Software Reset Register (Address at PLB_base_address + 0x2828)**

Bit Number	Default	Access	Description
0	0	WO	Transmitter path reset. When written with a '1', forces the entire transmitter path of the core to be reset. This also asserts the `tx_reset` signal of Table 5-1. This reset does not affect transmitter configuration settings. If read, always returns 0.
1	0	WO	Receiver path reset. When written with a '1', forces the entire receiver path of the core to be reset. This also asserts the `rx_reset` signal of Table 5-1. This reset does not affect receiver configuration settings. If read, always returns 0.
2	0	WO	PTP Transmitter logic reset. When written with a '1', forces the PTP transmitter logic of the core to be reset. This is a subset of the full transmitter path reset of bit 0. This reset does not affect PTP transmitter configuration settings. If read, always returns 0.

Table 10-15: **Software Reset Register (Address at PLB_base_address + 0x2828)**

Bit Number	Default	Access	Description
3	0	WO	PTP Receiver logic reset. When written with a '1', forces the PTP receiver logic of the core to be reset. This is a subset of the full receiver path reset of bit 1. This reset does not affect PTP receiver configuration settings. If read, always returns 0.
31-4	0	RO	Unused

MAC Header Filter Configuration

When the core is generated in "EDK pcore Format", the "Legacy MAC Header Filters" are not included since the xps_ll_temac can optionally contain its own Address Filter logic. When not provided, the following address locations will return 0s for a read and all writes will be ignored.

When the core is generated in "Standard CORE Generator Format", the "Legacy MAC Header Filters" are provided. These filters are present on the Rx Legacy traffic path, are capable of providing match recognition logic against eight unique MAC frame headers. Each of the eight individual filters require eight memory mapped registers to configure them, as defined in Table 10-16. Each individual filter contains its own set of these eight registers. When interpreting Table 10-16, the variable *filter#* should be replaced with an integer number between 0 and 7, which represent the eight individual filters.

Table 10-16: **MAC Header Filter Configuration Registers**

Address	Default	Access	Description
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x0	0xFFFFFFFF	R/W	Match Pattern: Ethernet frame bits 0 to 31 32 bit pattern to match against the Ethernet frame bits 0 to 31. Specifically, match pattern bits: [31:0]: MAC Destination Address Field bits [31:0]
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x4	0x0000FFFF	R/W	Match Pattern: Ethernet frame bits 32 to 63 32 bit pattern to match against the Ethernet frame bits 32 to 63. Specifically, match pattern bits: [15:0]: MAC Destination Address Field bits [47:32] [31:16]: MAC Source Address Field bits [15:0]
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x8	0x00000000	R/W	Match Pattern: Ethernet frame bits 64 to 95 32 bit pattern to match against the Ethernet frame bits 64 to 95. Specifically, match pattern bits: [31:0]: MAC Source Address bits [47:16]

Table 10-16: **MAC Header Filter Configuration Registers *(Cont'd)***

Address	Default	Access	Description
PLB_base_address + 0x3000 + (filter# * 0x20) + 0xC	0x00000000	R/W	Match Pattern: Ethernet frame bits 96 to 127 32 bit pattern to match against the Ethernet frame bits 96 to 127. For frames with a VLAN tag, match pattern bits[31:0] can be matched against the full VLAN field. For frames without a VLAN, match pattern bits[15:0] can be matched against the Length/Type field.
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x10	0xFFFFFFFF	R/W	Match Enable: Ethernet frame bits 0 to 31 There is a 1-to-1 correspondence between all bits in this register and all bits in the "Match Pattern: Ethernet frame bits 0 to 31" register. For each bit: logic 1 enables the match: the corresponding bit in the Match Pattern will be compared logic 0 disables the match: the corresponding bit in the Match Pattern will be a don't-care.
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x14	0x0000FFFF	R/W	Match Enable: Ethernet frame bits 32 to 63 There is a 1-to-1 correspondence between all bits in this register and all bits in the "Match Pattern: Ethernet frame bits 32 to 63" register. For each bit: logic 1 enables the match: the corresponding bit in the Match Pattern will be compared logic 0 disables the match: the corresponding bit in the Match Pattern will be a don't-care.
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x18	0x00000000	R/W	Match Enable: Ethernet frame bits 64 to 95 There is a 1-to-1 correspondence between all bits in this register and all bits in the "Match Pattern: Ethernet frame bits 64 to 95" register. For each bit: logic 1 enables the match: the corresponding bit in the Match Pattern will be compared logic 0 disables the match: the corresponding bit in the Match Pattern will be a don't-care.
PLB_base_address + 0x3000 + (filter# * 0x20) + 0x1C	0x00000000	R/W	Match Enable: Ethernet frame bits 96 to 127 There is a 1-to-1 correspondence between all bits in this register and all bits in the "Match Pattern: Ethernet frame bits 96 to 127" register. For each bit: logic 1 enables the match: the corresponding bit in the Match Pattern will be compared logic 0 disables the match: the corresponding bit in the Match Pattern will be a don't-care.

Tri-Mode Ethernet MAC Address Space

When the core is generated in "EDK pcore Format" for import into EDK and connection to the xps_ll_temac, the address space defined in this section is not included and the address space will return 0s for a read and all writes will be ignored.

When the core is generated in "Standard CORE Generator Format", the address space of the Ethernet MAC is incorporated into the address space of the Ethernet AVB Endpoint core as illustrated in Figure 10-3. The Ethernet MAC Address space is then split into two sections:

- "MAC Configuration and Statistics"
- "MAC MDIO Registers"

MAC Configuration and Statistics

Table 10-17 defines the statistic registers and configuration registers of the Tri-Mode Ethernet MAC core. These are listed with their assigned addresses. See the Tri-Mode Ethernet MAC User Guide (UG138) and the Ethernet Statistics User Guide (UG170) for additional descriptions of these registers.

Table 10-17: **Tri-Mode Ethernet MAC and Ethernet Statistics Configuration Registers**

Address	Description
(PLB_base_address + 0x4000) to (PLB_base_address + 0x41FF)	A maximum of 64 configurable Ethernet MAC statistics registers can be accessed through the PLB interface (let the statistics registers be numbered by STATISTIC_NUMBER, from 0 to 63). Each statistic returns a 64-bit counter value. Accordingly: Address of STATISTIC_NUMBER = (PLB_base_address + 0x4000 + [STATISTIC_NUMBER * 8])
PLB_base_address + 0x5000	Receiver Configuration (Word 0)
PLB_base_address + 0x5200	Receiver Configuration (Word 1)
PLB_base_address + 0x5400	Transmitter Configuration
PLB_base_address + 0x5600	Flow Control Configuration
PLB_base_address + 0x5800	MAC Speed Configuration
PLB_base_address + 0x5A00	Management Configuration

MAC Address Filter Registers

The Address Filter, optionally present in the Tri-Mode Ethernet MAC LogiCORE™ IP solution, must not used. Instead, new"Legacy MAC Header Filters" have been added to the Receiver Legacy Traffic path, which is capable of providing address recognition for eight unique MAC addresses. See "MAC Header Filter Configuration."

MAC MDIO Registers

The Tri-Mode Ethernet MAC has MDIO master capability. To access an MDIO register via the Ethernet MAC, construct the address as follows:

```
MDIO register address = PLB_base_address + 0x6000 + (MDIO_ADDRESS *8)
```

where `MDIO_ADDRESS` is a 10-bit binary address, constructed from the 5-bit MDIO Physical Address (PHYAD) and the 5-bit MDIO Register Address (REGAD) as follows:

```
MDIO_ADDRESS <= {PHYAD, REGAD}
```

See the *Tri-Mode Ethernet MAC User Guide* and IEEE802.3 for further MDIO information.

XILINX®

Chapter 11

Constraining the Core

This chapter defines the Ethernet AVB Endpoint core constraints. An example user constraints file (UCF) is provided for the core and the HDL example design.

Required Constraints

Device, Package, and Speedgrade Selection

The Ethernet AVB Endpoint core can be implemented in Spartan®-3, Spartan-3E, Spartan-3A/3A DSP, Spartan-6, Virtex®-5 and Virtex-6 devices that are large enough to accommodate the core, and meet the following speed grades:

- -1 for Virtex-5 and Virtex-6 devices
- -2 for Spartan-6 devices
- -4 for all Spartan-3 devices

I/O Location Constraints

No specific I/O location constraints are required.

Placement Constraints

No specific placement constraints are required.

Timing Constraints

The core can have up to five separate clock domains:

- `plb_clk` for the main EDK PLB and processor clock frequency
- `host_clk` for the management interface logic of the connected Tri-Mode Ethernet MAC
- `tx_clk` for the MAC transmitter clock domain
- `rx_clk` for the MAC receiver clock domain
- `rtc_clk` for the"Real Time Clock" reference frequency

These clock nets and the signals within the core that cross these clock domains must be constrained appropriately in a UCF.

Sections of UCF syntax are used in the following descriptions to provide examples.

PERIOD Constraints for Clock Nets

PLB_clk

The clock provided to `PLB_clk` must be constrained to the appropriate frequency. Note the frequency range of the embedded processor to which this bus is connected. For example, the maximum clock speed of the MicroBlaze™ processor is 100 MHz.

The following UCF syntax shows a 100 MHz period constraint being applied to the `PLB_clk` signal:

```
NET "plb_clk" TNM_NET    = "plb_clk";
TIMEGRP "plb_clock"      = "plb_clk";
TIMESPEC "TS_plb_clock" = PERIOD "plb_clock" 10000 ps HIGH 50 %;
```

host_clk

The clock provided to `host_clk` must be constrained to the desired Management Interface operating frequency of the Tri-Mode Ethernet MAC. If `host_clk` is connected to the same clock source as any other Ethernet AVB Endpoint input clock (for example `PLB_clk` or `ref_clk`), then this constraint is unnecessary and can be removed.

The maximum supported frequency of `host_clk`, as specified by the Tri-Mode Ethernet MAC core, is 125 MHz.

The following UCF syntax shows a 125 MHz period constraint being applied to `host_clk`:

```
NET "host_clk" TNM_NET = "host_clk";
TIMEGRP "host_clock" = "host_clk";
TIMESPEC "TS_host_clock" = PERIOD "host_clock" 8000 ps HIGH 50
%;
```

tx_clk

The interface clock of the Ethernet MACs transmitter must be constrained to the correct maximum frequency. This is 125 MHz for 1-Gigabit Ethernet rates.

The following UCF syntax shows the necessary constraints being applied to `tx_clk`:

```
NET "tx_clk" TNM_NET = "tx_clk";
TIMEGRP "tx_clock" = "tx_clk";
TIMESPEC "TS_tx_clock" = PERIOD "tx_clock" 8000 ps HIGH 50 %;
```

rx_clk

The interface clock of the Ethernet MACs receiver must be constrained to the correct maximum frequency. This is 125 MHz for 1-Gigabit Ethernet rates.

The following UCF syntax shows the necessary constraints being applied to `rx_clk`:

```
NET "rx_clk" TNM_NET = "rx_clk";
TIMEGRP "rx_clock" = "rx_clk";
TIMESPEC "TS_rx_clock" = PERIOD "rx_clock" 8000 ps HIGH 50 %;
```

rtc_clk

The RTC can be incremented from any available clock frequency that is greater than the AVB standards defined minimum of 25 MHz. However, the faster the frequency of the clock, the smaller will be the step increment and the smoother will be the overall RTC increment rate. Xilinx recommends clocking the RTC logic at 125 MHz because this is a readily available clock source (obtained from the transmit clock source of the Ethernet MAC at 1 Gbps speed). This frequency significantly exceeds the minimum performance of the *P802.1AS* specification.

The following UCF syntax shows a 125 MHz period constraint being applied to `rtc_clk`:

```
NET "rtc_clk" TNM_NET   = "rtc_clk";
TIMEGRP "rtc_clock"     = "rtc_clk";
TIMESPEC "TS_rtc_clock" = PERIOD "rtc_clock" 8000 ps HIGH 50 %;
```

Timespecs for Critical Logic within the Core

Signals must cross clock domains at certain points within the core. To guarantee that these signals are sampled correctly on the new clock domain, many constraints are required, and must *not* be removed. These constraints are also present in the example design UCF delivered with the core.

```
##############################################################
# Clock Domain Crossing Constraints                          #
##############################################################


# clock domain crossing constraints for Tx timestamp logic
#------------------------------------------------------------

INST "*top/tx_rtc_sample_inst/sample_toggle_req" TNM = FFS
"tx_sample_req";
INST "*top/tx_rtc_sample_inst/resync_sample_toggle_req/data_sync"
TNM = FFS "tx_sample_req_resync";
TIMESPEC "ts_tx_sample_req" = FROM "tx_sample_req" TO
"tx_sample_req_resync" 6.5 ns DATAPATHONLY;

INST "*top/tx_rtc_sample_inst/sample_taken_toggle" TNM = FFS
"tx_sample_taken";
INST "*top/tx_rtc_sample_inst/resync_sample_taken_toggle/data_sync"
TNM = FFS "tx_sample_taken_resync";
TIMESPEC "ts_tx_sample_taken" = FROM "tx_sample_taken" TO
"tx_sample_taken_resync" TIG;

INST "*top/tx_rtc_sample_inst/timestamp*" TNM = FFS "tx_timestamp";
TIMESPEC "ts_tx_timestamp_route" = FROM "tx_timestamp" TO "FFS"  8 ns
DATAPATHONLY;


# clock domain crossing constraints for Rx timestamp logic
#------------------------------------------------------------

INST "*top/rx_rtc_sample_inst/sample_toggle_req" TNM = FFS
"rx_sample_req";
INST "*top/rx_rtc_sample_inst/resync_sample_toggle_req/data_sync"
TNM = FFS "rx_sample_req_resync";
TIMESPEC "ts_rx_sample_req" = FROM "rx_sample_req" TO
"rx_sample_req_resync" 6.5 ns DATAPATHONLY;
```

```
INST "*top/rx_rtc_sample_inst/sample_taken_toggle" TNM = FFS
"rx_sample_taken";
INST "*top/rx_rtc_sample_inst/resync_sample_taken_toggle/data_sync"
TNM = FFS "rx_sample_taken_resync";
TIMESPEC "ts_rx_sample_taken" = FROM "rx_sample_taken" TO
"rx_sample_taken_resync" TIG;

INST "*top/rx_rtc_sample_inst/timestamp*" TNM = FFS "rx_timestamp";
TIMESPEC "ts_rx_timestamp_route" = FROM "rx_timestamp" TO "FFS"  8 ns
DATAPATHONLY;


# clock domain crossing constraints for Rx PTP Packet Buffer logic
#-------------------------------------------------------------

INST
"*top/ptp_packet_buffer_inst/rx_ptp_packet_buffer_inst/rx_mac_logic_in
st/rx_clear_toggle" TNM = FFS "rx_clear_toggle";
INST
"*top/ptp_packet_buffer_inst/rx_ptp_packet_buffer_inst/rx_mac_logic_in
st/resync_clear_toggle/data_sync" TNM = FFS "rx_clear_toggle_resync";
TIMESPEC "ts_rx_clear_toggle" = FROM "rx_clear_toggle" TO
"rx_clear_toggle_resync" TIG;

INST
"*top/ptp_packet_buffer_inst/rx_ptp_packet_buffer_inst/rx_mac_logic_in
st/address*" TNM = FFS "rx_buf_addr";
INST
"*top/ptp_packet_buffer_inst/rx_ptp_packet_buffer_inst/rx_mac_logic_in
st/rx_packet*" TNM = FFS "rx_buf_addr_sample";
TIMESPEC "ts_rx_buf_addr" = FROM "rx_buf_addr" TO "rx_buf_addr_sample"
64 ns DATAPATHONLY;


# clock domain crossing constraints for Tx PTP Packet Buffer logic
#-------------------------------------------------------------

INST
"*top/ptp_packet_buffer_inst/tx_ptp_packet_buffer_inst/tx_mac_logic_in
st/tx_valid_reg2" TNM = FFS "tx_valid_reg2";
INST
"*top/ptp_packet_buffer_inst/tx_ptp_packet_buffer_inst/tx_mac_logic_in
st/resync_frame_tx_toggle/data_sync" TNM = FFS "tx_valid_reg2_resync";
TIMESPEC "ts_tx_valid_reg2" = FROM "tx_valid_reg2" TO
"tx_valid_reg2_resync" TIG;


# clock domain crossing constraints for Rx Configuration
#-------------------------------------------------------------

INST "*top/avb_configuration_inst/promiscuous_mode_int" TNM = FFS
"promiscuous_mode";
INST
"*top/legacy_inst*address_filter_inst/*resync_promiscuous_mode/data_sy
nc" TNM = FFS "promiscuous_mode_resync";
TIMESPEC "ts_promiscuous_mode" = FROM "promiscuous_mode" TO
"promiscuous_mode_resync" TIG;
```

```
INST "*top/avb_configuration_inst/vlan_priority_a_int*" TNM = FFS
"vlan_priority_a";
INST "*top/rx_splitter_inst/vlan_priority_a_sample*" TNM = FFS
"vlan_priority_a_sample";
TIMESPEC "ts_vlan_priority_a_sample" = FROM "vlan_priority_a" TO
"vlan_priority_a_sample" TIG;

INST "*top/avb_configuration_inst/vlan_priority_b_int*" TNM = FFS
"vlan_priority_b";
INST "*top/rx_splitter_inst/vlan_priority_b_sample*" TNM = FFS
"vlan_priority_b_sample";
TIMESPEC "ts_vlan_priority_b_sample" = FROM "vlan_priority_b" TO
"vlan_priority_b_sample" TIG;


# clock domain crossing constraints for Tx Configuration
#------------------------------------------------------------

INST "*top/avb_configuration_inst/tx_cpu_reclock/wr_toggle"
TNM = FFS "tx_wr_toggle";
INST
"*top/avb_configuration_inst/tx_cpu_reclock/resync_write_toggle/data_s
ync"  TNM = FFS "resync_tx_write_toggle";
TIMESPEC "ts_tx_wr_toggle" = FROM "tx_wr_toggle" TO
"resync_tx_write_toggle" TIG;

INST "*top/avb_configuration_inst/tx_cpu_reclock/rd_toggle"
TNM = FFS "tx_rd_toggle";
INST
"*top/avb_configuration_inst/tx_cpu_reclock/resync_read_toggle/data_sy
nc"   TNM = FFS "resync_tx_read_toggle";
TIMESPEC "ts_tx_rd_toggle" = FROM "tx_rd_toggle" TO
"resync_tx_read_toggle" TIG;

INST "*top/avb_configuration_inst/tx_cpu_reclock/new_rd_toggle" TNM =
FFS "cpu_tx_rd_toggle";
INST
"*top/avb_configuration_inst/tx_cpu_reclock/resync_new_rd_toggle/data_
sync" TNM = FFS "resync_cpu_tx_rd_toggle";
TIMESPEC "ts_cpu_tx_rd_toggle" = FROM "cpu_tx_rd_toggle" TO
"resync_cpu_tx_rd_toggle" TIG;

INST "*top/avb_configuration_inst/tx_cpu_reclock/new_wr_toggle" TNM =
FFS "cpu_tx_wr_toggle";
INST
"*top/avb_configuration_inst/tx_cpu_reclock/resync_new_wr_toggle/data_
sync" TNM = FFS "resync_cpu_tx_wr_toggle";
TIMESPEC "ts_cpu_tx_wr_toggle" = FROM "cpu_tx_wr_toggle" TO
"resync_cpu_tx_wr_toggle" TIG;

INST "*top/avb_configuration_inst/tx_cpu_reclock/new_be*"   TNM = FFS
"tx_cpu_sample";
INST "*top/avb_configuration_inst/tx_cpu_reclock/new_addr*" TNM = FFS
"tx_cpu_sample";
TIMESPEC "ts_tx_cpu_sample" = FROM "cpu_bus" TO "tx_cpu_sample" 16 ns
DATAPATHONLY;

INST "*top/avb_configuration_inst/clear_tx_int" TNM = FFS
"tx_regs_sample";
```

```
INST "*top/avb_configuration_inst/tx_send_frame*" TNM = FFS
"tx_regs_sample";
INST "*top/avb_configuration_inst/tx_sendslope_int*" TNM = FFS
"tx_regs_sample";
INST "*top/avb_configuration_inst/tx_idleslope_int*" TNM = FFS
"tx_regs_sample";
TIMESPEC "ts_tx_regs_sample" = FROM "cpu_bus" TO "tx_regs_sample" 24 ns
DATAPATHONLY;

INST "*top/avb_configuration_inst/rd_data_tx*" TNM = FFS "tx_rd_data";
INST "*top/avb_configuration_inst/cpu_rd_data*" TNM = FFS
"tx_cpu_rd_data";
TIMESPEC "ts_tx_rd_data" = FROM "tx_rd_data" TO "tx_cpu_rd_data" 16 ns
DATAPATHONLY;


# clock domain crossing constraints for RTC Configuration Logic
#-------------------------------------------------------------

INST"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/wr_toggle"
TNM = FFS "rtc_wr_toggle";
INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/resync_write_tog
gle/data_sync"  TNM = FFS "resync_rtc_write_toggle";
TIMESPEC "ts_rtc_wr_toggle" = FROM "rtc_wr_toggle" TO
"resync_rtc_write_toggle" TIG;

INST"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/rd_toggle"
TNM = FFS "rtc_rd_toggle";
INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/resync_read_togg
le/data_sync"   TNM = FFS "resync_rtc_read_toggle";
TIMESPEC "ts_rtc_rd_toggle" = FROM "rtc_rd_toggle" TO
"resync_rtc_read_toggle" TIG;

INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/new_rd_toggle"
TNM = FFS "cpu_rtc_rd_toggle";
INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/resync_new_rd_to
ggle/data_sync" TNM = FFS "resync_cpu_rtc_rd_toggle";
TIMESPEC "ts_cpu_rtc_rd_toggle" = FROM "cpu_rtc_rd_toggle" TO
"resync_cpu_rtc_rd_toggle" TIG;

INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/new_wr_toggle"
TNM = FFS "cpu_rtc_wr_toggle";
INST
"*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/resync_new_wr_to
ggle/data_sync" TNM = FFS "resync_cpu_rtc_wr_toggle";
TIMESPEC "ts_cpu_rtc_wr_toggle" = FROM "cpu_rtc_wr_toggle" TO
"resync_cpu_rtc_wr_toggle" TIG;

INST "*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/new_be*"
TNM = FFS "rtc_cpu_sample";
INST "*top/rtc_inst/rtc_configuration_inst/rtc_cpu_reclock/new_addr*"
TNM = FFS "rtc_cpu_sample";
TIMESPEC "ts_rtc_cpu_sample" = FROM "cpu_bus" TO "rtc_cpu_sample" 16 ns
DATAPATHONLY;
```

```
INST "*top/rtc_inst/rtc_configuration_inst/reg_nanosec_offset*" TNM =
FFS "rtc_regs_sample";
INST "*top/rtc_inst/rtc_configuration_inst/reg_sec_offset*" TNM = FFS
"rtc_regs_sample";
INST "*top/rtc_inst/rtc_configuration_inst/reg_epoch_offset*" TNM =
FFS "rtc_regs_sample";
INST "*top/rtc_inst/rtc_configuration_inst/reg_rtc_increment*" TNM =
FFS "rtc_regs_sample";
INST "*top/rtc_inst/rtc_configuration_inst/reg_offset_8k*" TNM = FFS
"rtc_regs_sample";
TIMESPEC "ts_rtc_regs_sample" = FROM "cpu_bus" TO "rtc_regs_sample" 24
ns DATAPATHONLY;

INST "*top/rtc_inst/rtc_configuration_inst/rd_data_result*" TNM = FFS
"rtc_rd_data";
INST "*top/rtc_inst/rtc_configuration_inst/cpu_rd_data*" TNM = FFS
"rtc_cpu_rd_data";
TIMESPEC "ts_rtc_rd_data" = FROM "rtc_rd_data" TO "rtc_cpu_rd_data" 16
ns DATAPATHONLY;

INST "*top/rtc_inst/rtc_configuration_inst/pulse1div128sec_toggle" TNM
= FFS "pulse1div128sec_toggle";
INST
"*top/rtc_inst/rtc_configuration_inst/resync_set_toggle/data_sync"
TNM = FFS "resync_set_toggle";
TIMESPEC "ts_pulse1div128sec_toggle" = FROM "pulse1div128sec_toggle" TO
"resync_set_toggle" 8 ns DATAPATHONLY;


# clock domain crossing constraints for MAC Host I/F Logic
#-------------------------------------------------------------

INST "*top*generic_host_if_inst/wr_toggle" TNM = FFS "wr_toggle";
INST "*top*generic_host_if_inst/resync_write_toggle/data_sync" TNM =
FFS "resync_write_toggle";
TIMESPEC "ts_wr_toggle" = FROM "wr_toggle" TO "resync_write_toggle" 8
ns DATAPATHONLY;

INST "*top*generic_host_if_inst/rd_toggle" TNM = FFS "rd_toggle";
INST "*top*generic_host_if_inst/resync_read_toggle/data_sync" TNM =
FFS "resync_read_toggle";
TIMESPEC "ts_rd_toggle" = FROM "rd_toggle" TO "resync_read_toggle"  8
ns DATAPATHONLY;
INST "*top/include_plb.plb_intf_inst/Bus2IP_Addr*"  TNM = FFS
"cpu_bus";
INST "*top/include_plb.plb_intf_inst/Bus2IP_Data*" TNM = FFS "cpu_bus";
INST "*top/include_plb.plb_intf_inst/Bus2IP_BE*" TNM = FFS "cpu_bus";
INST "*top*generic_host_if_inst/host_address_bit10" TNM = FFS
"host_sample";
INST "*top*generic_host_if_inst/host_address*" TNM = FFS "host_sample";
INST "*top*generic_host_if_inst/stats_upper_word*" TNM = FFS
"host_sample";
INST "*top*generic_host_if_inst/host_wr_data*" TNM = FFS "host_sample";
INST "*top*generic_host_if_inst/host_be*" TNM = FFS "host_sample";
TIMESPEC "ts_host_sample" = FROM "cpu_bus" TO "host_sample" 8 ns
DATAPATHONLY;
```

```
INST "*top*generic_host_if_inst/host_toggle_reg2" TNM = FFS
"host_toggle";
INST "*top*generic_host_if_inst/resync_host_toggle/data_sync" TNM =
FFS "resync_host_toggle";
TIMESPEC "ts_host_toggle" = FROM "host_toggle" TO "resync_host_toggle"
8 ns DATAPATHONLY;

INST "*top*generic_host_if_inst/host_rd_data_result*" TNM = FFS
"host_rd_data";
INST "*top*generic_host_if_inst/cpu_rd_data*" TNM = FFS "cpu_rd_data";
TIMESPEC "ts_cpu_rd_data" = FROM "host_rd_data" TO "cpu_rd_data" 8 ns
DATAPATHONLY;
```

XILINX®

Chapter 12

System Integration

As described in Chapter 4, "Generating the Core" and Chapter 5, "Core Architecture", the core can be generated in one of two formats:

- "Standard CORE Generator Format"

  This option will deliver the core in the standard CORE Generator™ output format, as used by many other cores including previous versions of this core and all other Ethernet LogiCORE™ IP solutions.

  When generated in this format, the core is designed to interface to the LogiCORE IP Tri-Mode Ethernet MAC or the LogiCORE IP Embedded Tri-Mode Ethernet MAC wrappers. Refer to "Using the Xilinx LogiCORE IP Tri-Mode Ethernet MACs" section of this chapter.

- "EDK pcore Format"

  This option will deliver the core in the standard pcore format, suitable for directly importing into the Xilinx Embedded Development Kit (EDK) environment.

  When generated in this format, the core is designed to interface to the XPS LocalLink Tri-Mode Ethernet MAC (xps_ll_temac). Refer to "Using the Xilinx XPS LocalLink Tri-Mode Ethernet MAC" section of this chapter.

Using the Xilinx LogiCORE IP Tri-Mode Ethernet MACs

The Ethernet AVB Endpoint core should be generated in the "Standard CORE Generator Format".

The Ethernet AVB Endpoint core can be connected to the following Ethernet MACs from the CORE Generator LogiCORE IP library:

- "LogiCORE IP Tri-Mode Ethernet MAC (Soft Core)", available for all Spartan®-3, Spartan-3E, Spartan-3A, Spartan-3A DSP, Spartan-6, Virtex®-5 and Virtex-6 devices.
- "LogiCORE IP Embedded Tri-Mode Ethernet MACs", available in selected Virtex-5 and Virtex-6 devices.

Please also refer to individual product documentation.

LogiCORE IP Tri-Mode Ethernet MAC (Soft Core)

Tri-Mode Ethernet MAC Core Generation

When generating the Tri-Mode Ethernet MAC (TEMAC) core in the CORE Generator software, be sure that the following options are selected:

- **Management Interface**. Enabled
- **Clock Enables**. Enabled
- **Address Filter**. Disabled

See the *Tri-Mode Ethernet MAC User Guide (UG138)* for additional information.

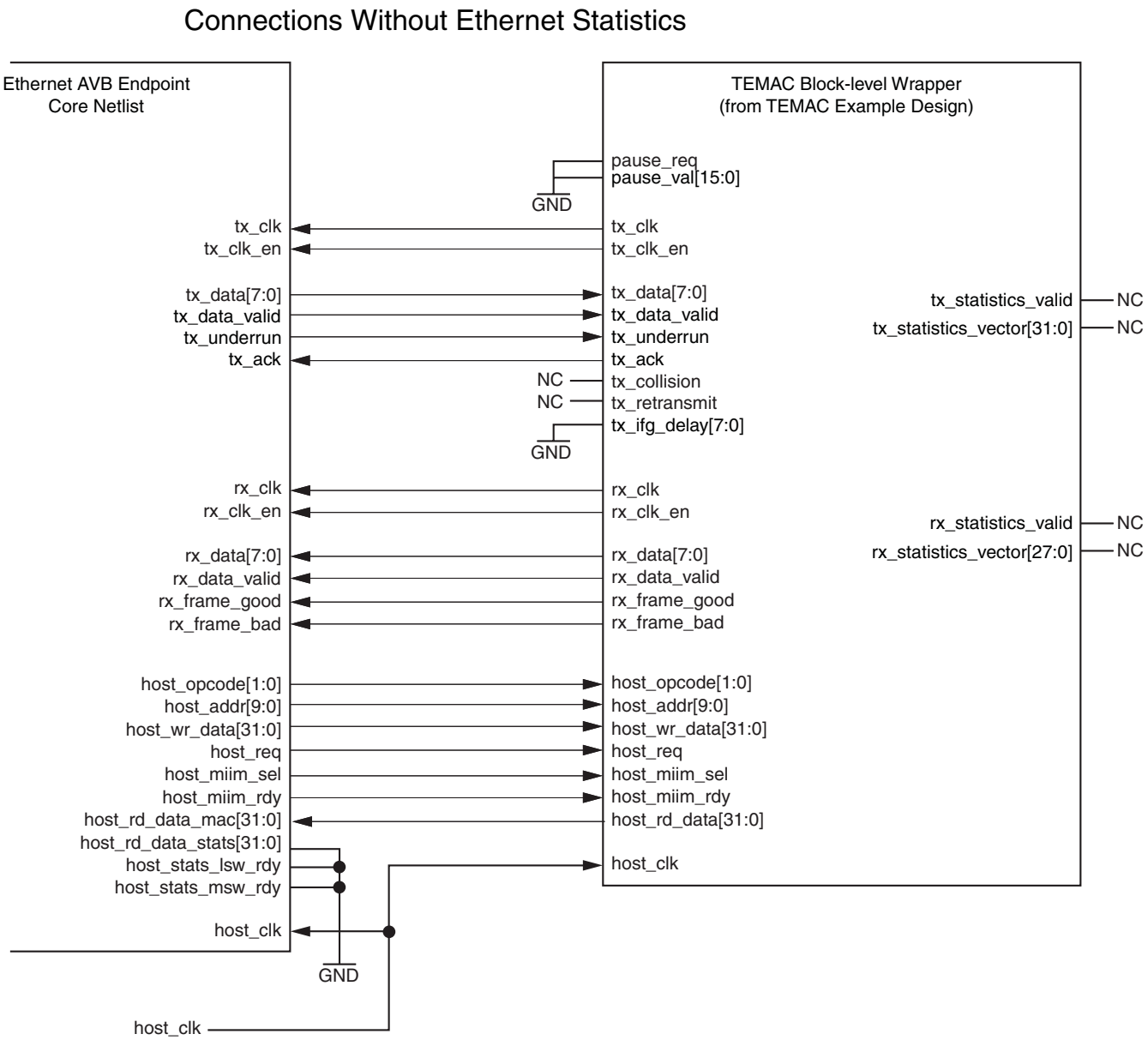

Figure 12-1: **Connection to the Tri-Mode Ethernet MAC Core (without Ethernet Statistics)**

Figure 12-1 illustrates the connection of the Ethernet AVB Endpoint core to the Xilinx Tri-Mode Ethernet MAC (TEMAC) core when not using the Ethernet Statistics core. Figure 12-1 provides detail for the connections between the two cores which were shown in Figure 5-1.

All connections, as shown, are logic-less connections. Because the AVB standard does not include support for half-duplex or flow control operation, the relevant half-duplex/flow-control signals of the TEMAC can be left unused: inputs can be tied to logic 0, outputs can be left unconnected.

Because the TEMAC core can often be used in different clocking modes, note the following:

- The Ethernet transmitter client clock domain must always be connected to the `tx_clk` input of the Ethernet AVB Endpoint core. Additionally, the transmitter clock enable, as used with the TEMAC, must always be connected to the `tx_clk_en` input of the Ethernet AVB Endpoint core.
- The Ethernet receiver client clock domain must always be connected to the `rx_clk` input of the Ethernet AVB Endpoint core. Additionally, the receiver clock enable, as used with the TEMAC, must always be connected to the `rx_clk_en` input of the Ethernet AVB Endpoint core.
- The `host_clk` inputs of the Ethernet AVB Endpoint and of the TEMAC must always share the same clock source. If desired, this can also be the clock source used for the PLB interface.

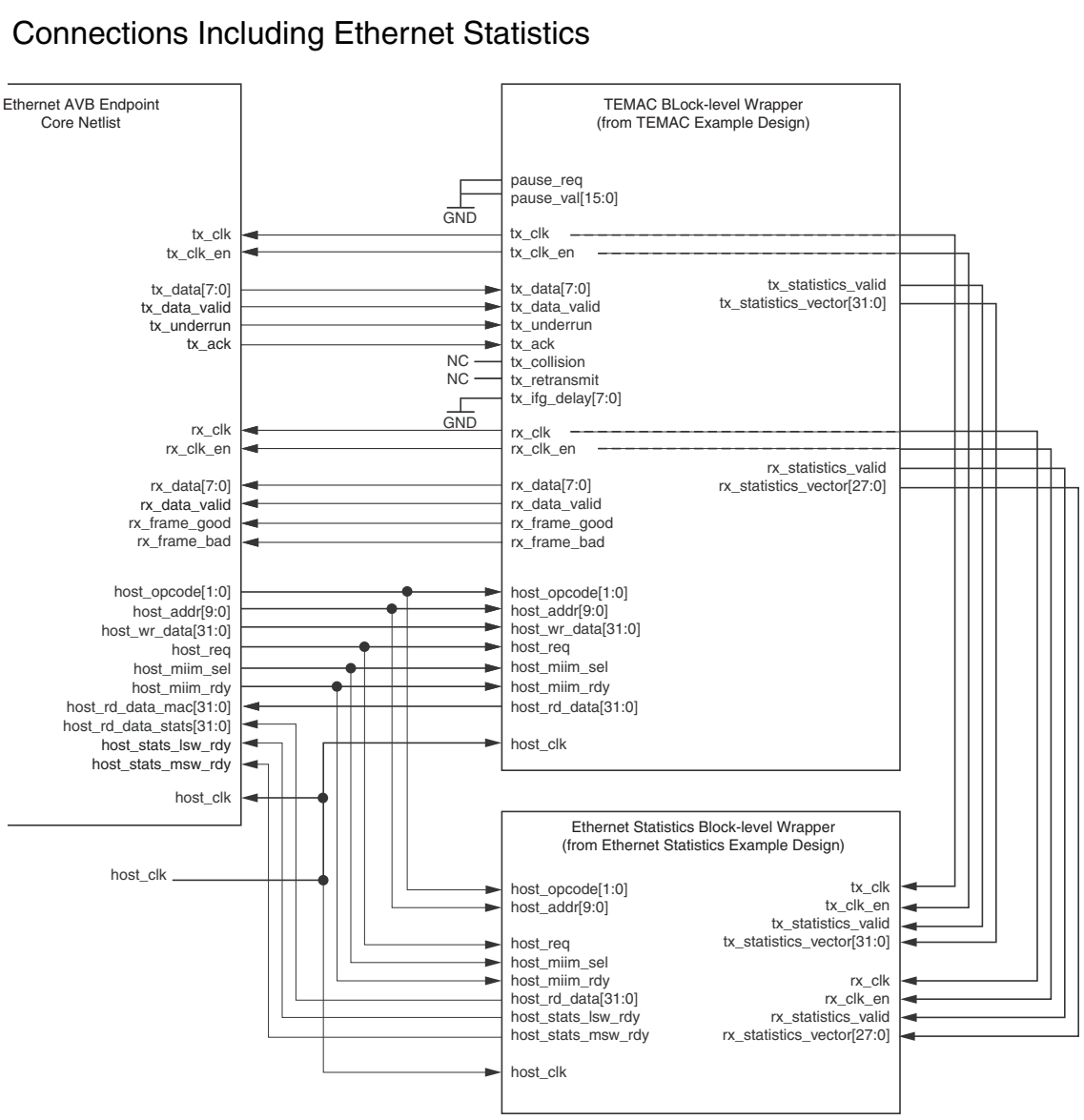

Figure 12-2: **Connection to the Tri-Mode Ethernet MAC and Ethernet Statistic Cores**

Figure 12-2 illustrates the connection of the Ethernet AVB Endpoint core to the Xilinx Tri-Mode Ethernet MAC (TEMAC) core when using the Ethernet Statistics core. This shares much in common with Figure 12-1, but take note of the following additional points:

- All the "MAC Management Interface" output signals of the Ethernet AVB Endpoint core connect directly to the signals of the same name at both the TEMAC and Ethernet Statistics cores.
- The Ethernet AVB Endpoint core provides two separate "MAC Management Interface" inputs for management reads. This allows for logic-less connections between all three cores as illustrated. To achieve this
  - connect `host_rd_data_mac[31:0]` of the Ethernet AVB Endpoint core to the `host_rd_data[31:0]` port of the TEMAC.
  - connect `host_rd_data_stats[31:0]` of the Ethernet AVB Endpoint core to the `host_rd_data[31:0]` port of the Ethernet Statistics core.

LogiCORE IP Embedded Tri-Mode Ethernet MACs

Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC Wrapper Generation

When generating the Virtex-5 FPGA Embedded Ethernet MAC Wrapper (EMAC) in the CORE Generator software, be sure that the following options are selected:

- Enable EMACs. Enable only a single EMAC (from the pair) at this time
- **Host Type**. Select *Host*
- **Speed.** Select *Tri-speed*
- **Global Buffer Usage**. *Clock Enable*
- **Flow Control Configuration.** Disabled
- **EMAC0 Configuration**. Enable *VLAN Enable* in both the *Transmitter Configuration* and *Receiver Configuration* boxes

See the *Virtex-5 Embedded Tri-Mode Ethernet MAC Wrapper Getting Started Guide (*UG340*)* for additional information.

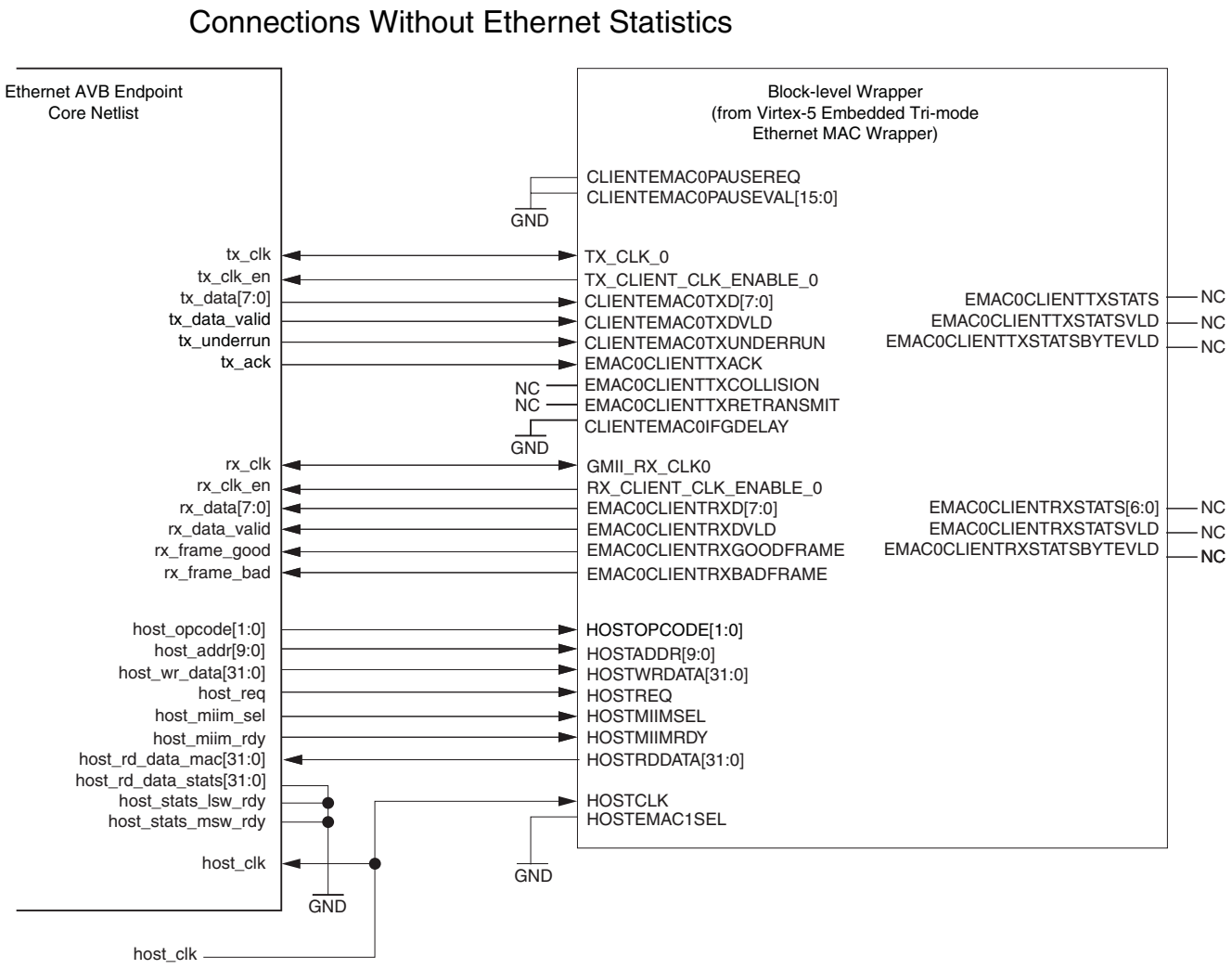

Figure 12-3: **Connection to the Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC (without Ethernet Statistics)**

Figure 12-3 illustrates the connection of the Ethernet AVB Endpoint core to the Xilinx Tri-Mode Ethernet MAC (EMAC) core when not using the Ethernet Statistics core. Figure 12-3 provides detail for the connections between the two cores which were shown in Figure 5-1.

All connections, as shown, are logic-less connections. Because the AVB standard does not include support for half-duplex or flow control operation, the relevant half-duplex/flow-control signals of the EMAC can be left unused: inputs can be tied to logic 0, outputs can be left unconnected.

Because the EMAC core can often be used in different clocking modes, note the following:

- The Ethernet transmitter client clock domain must always be connected to the `tx_clk` input of the Ethernet AVB Endpoint core. Additionally, the transmitter clock enable, as used with the EMAC, must always be connected to the `tx_clk_en` input of the Ethernet AVB Endpoint core.
- The Ethernet receiver client clock domain must always be connected to the `rx_clk` input of the Ethernet AVB Endpoint core. Additionally, the receiver clock enable, as used with the EMAC, must always be connected to the `rx_clk_en` input of the Ethernet AVB Endpoint core.
- The `host_clk` input of the Ethernet AVB Endpoint and the `HOSTCLK` input the EMAC must always share the same clock source.

Connections Including Ethernet Statistics

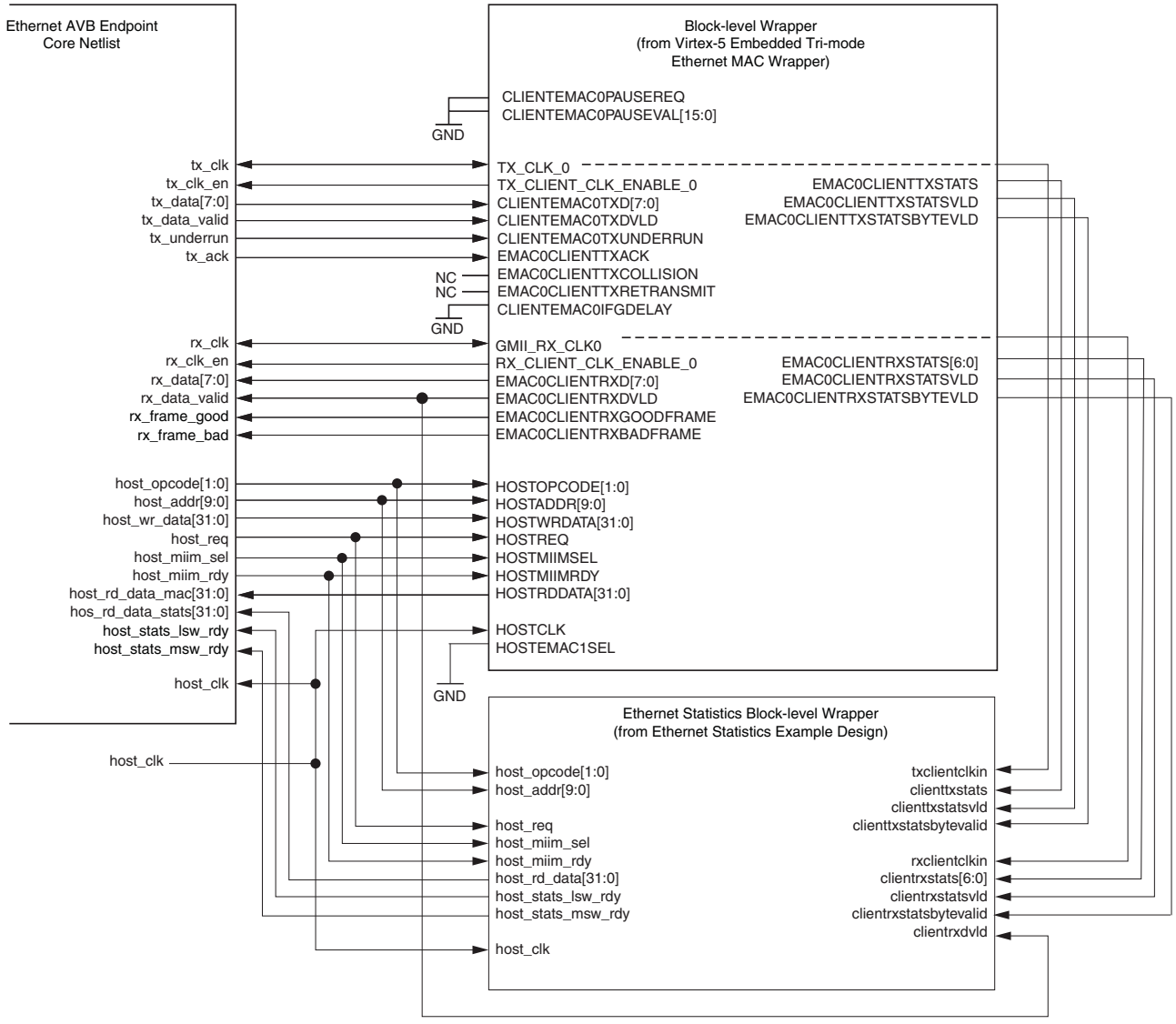

Figure 12-4: **Connection to the Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC and Ethernet Statistic Core**

Figure 12-4 illustrates the connection of the Ethernet AVB Endpoint core to the EMAC when using the Ethernet Statistics core. This shares much in common with Figure 12-2; however, note the following additional points:

- All of the "MAC Management Interface" output signals of the Ethernet AVB Endpoint core connect directly to the signals of both the EMAC and Ethernet Statistics cores.
- The Ethernet AVB Endpoint core provides two separate "MAC Management Interface" inputs for management reads. This allows for logic-less connections between all three cores as illustrated. To achieve this
  - connect `host_rd_data_mac[31:0]` of the Ethernet AVB Endpoint core to the `HOSTRDDATA[31:0]` port of the EMAC.

connect `host_rd_data_stats[31:0]` of the Ethernet AVB Endpoint core to the `host_rd_data[31:0]` port of the Ethernet Statistics core.

Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC

The Ethernet AVB Endpoint core will also connect directly to the Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC (EMAC). Use all of the preceding steps described for the Virtex-5 FPGA EMAC, the only difference being that Virtex-6 FPGA EMAC does not come in pairs; each EMAC is an individual element.

Connection of the PLB to the EDK for LogiCORE IP Ethernet MACs

Figure 12-5 illustrates the connection of the core to an embedded processor subsystem (MicroBlaze™ processor is illustrated). As shown:

- The PLB can be shared across all peripherals as illustrated.
- The "Interrupt Signals" should be connected to the inputs of an interrupt controller module, for example, the *xps_intc* core provided with the EDK.
- The embedded processor should be configured to use the software drivers provided with the core (see Chapter 13, "Software Drivers") (not illustrated).

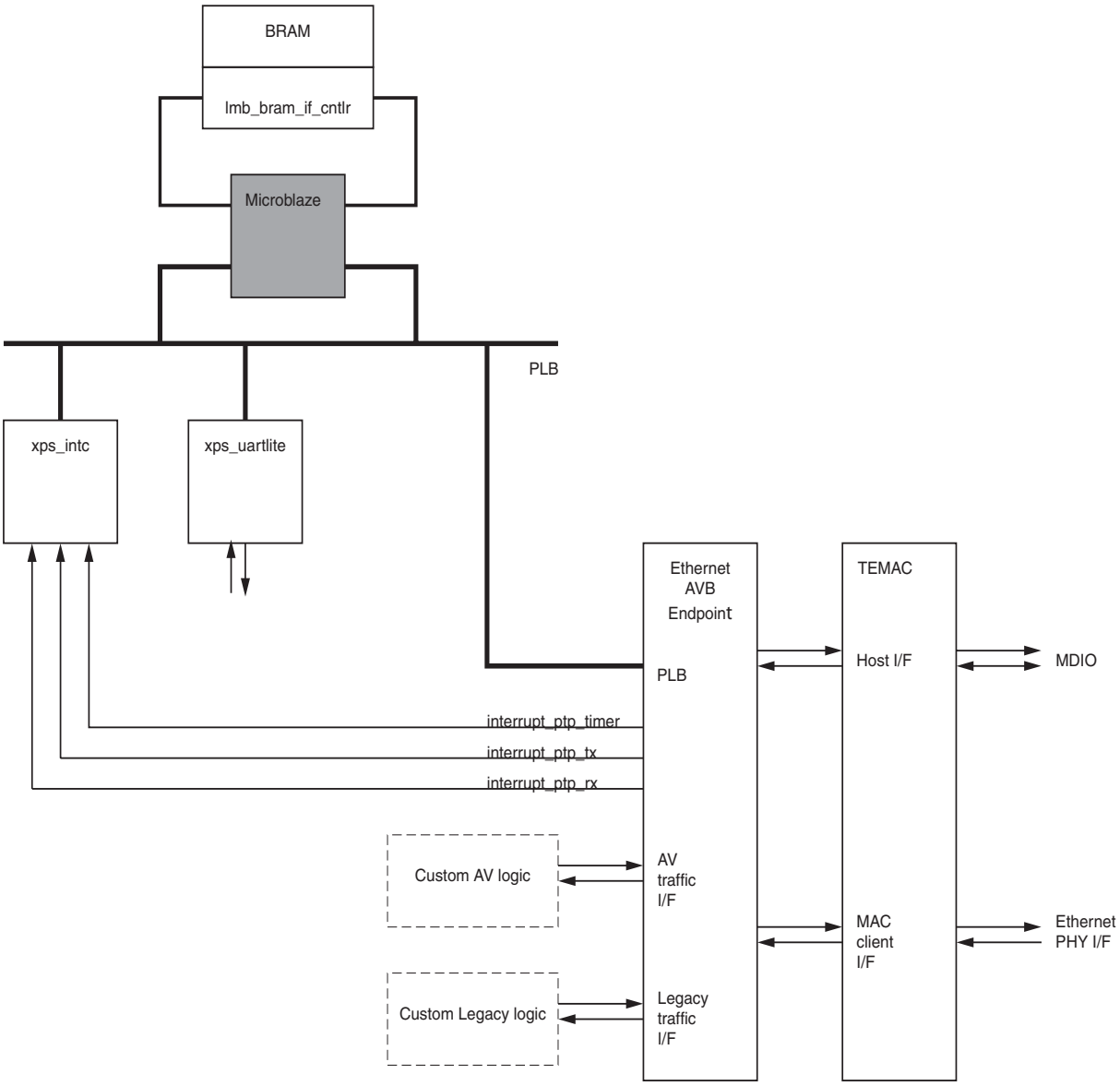

Figure 12-5: **Connection of the Ethernet AVB Endpoint Core into an Embedded Processor Sub-system**

Figure 12-5 can be implemented using the Xilinx tool set using two methods:

- "Using an EDK Project Top Level"
- "Using an ISE Software Top-Level Project"

Using an EDK Project Top Level

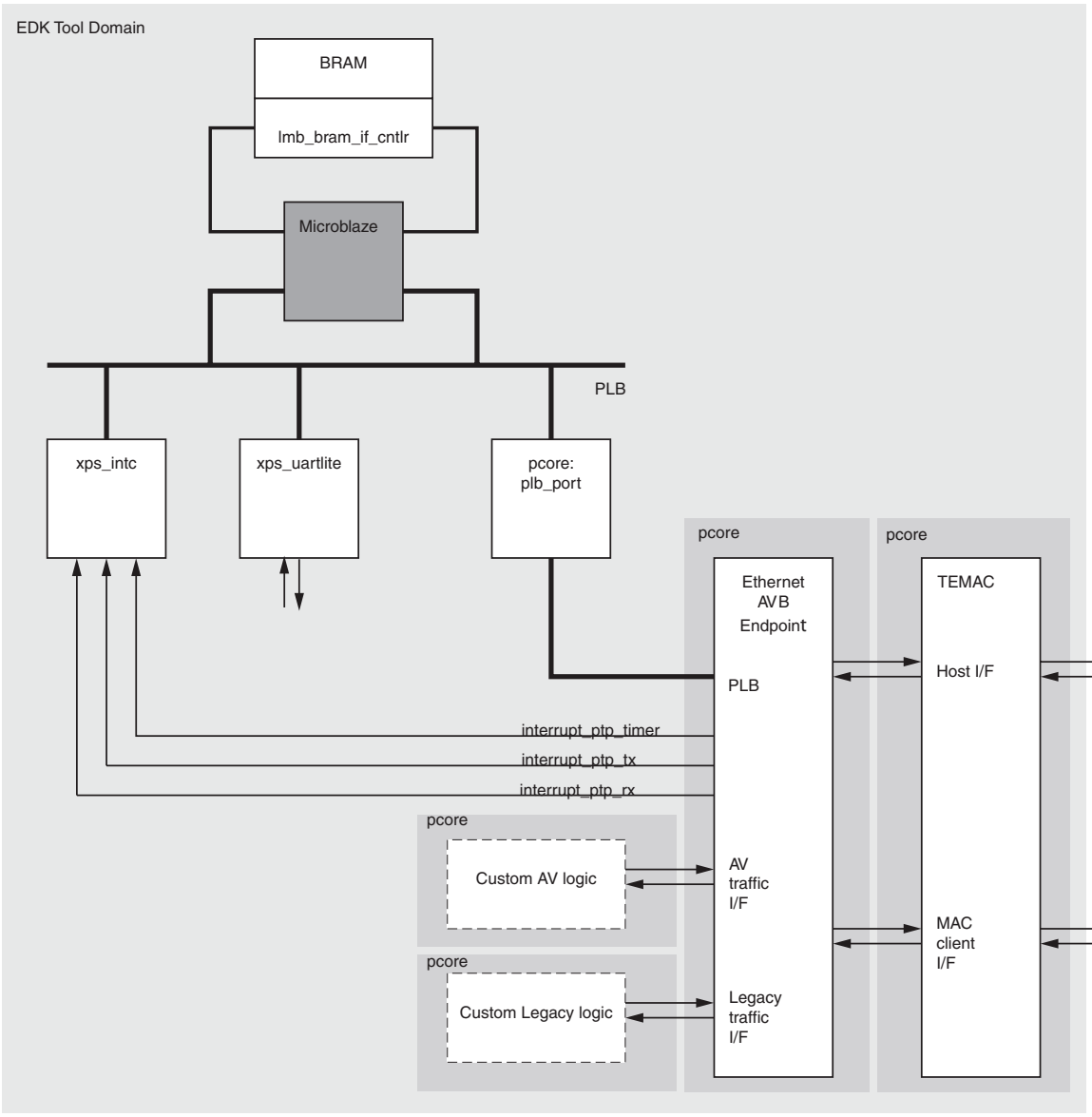

Figure 12-6: **Connection into an Embedded Processor Sub-system with an EDK Top-level Project**

Figure 12-6 shows the implementation using an EDK project. In this hierarchy, the Ethernet AVB Endpoint, Tri-Mode Ethernet MAC, and all custom logic blocks, must be manually translated into pcores using the standard pcore approach described in Xilinx Platform Studio documentation. The standard EDK flow can then be implemented to build the project.

In this example, the instance of the Ethernet AVB Endpoint core should be assigned a base address in the Microprocessor Hardware Specification (.mhs) file, to match that of the Ethernet AVB Endpoint "PLB Base Address" (in the generated netlist produced by the CORE Generator software). Then the AVB software drivers can be assigned to this instance in the Microprocessor Software Specification (.mss) file.

Using an ISE Software Top-Level Project

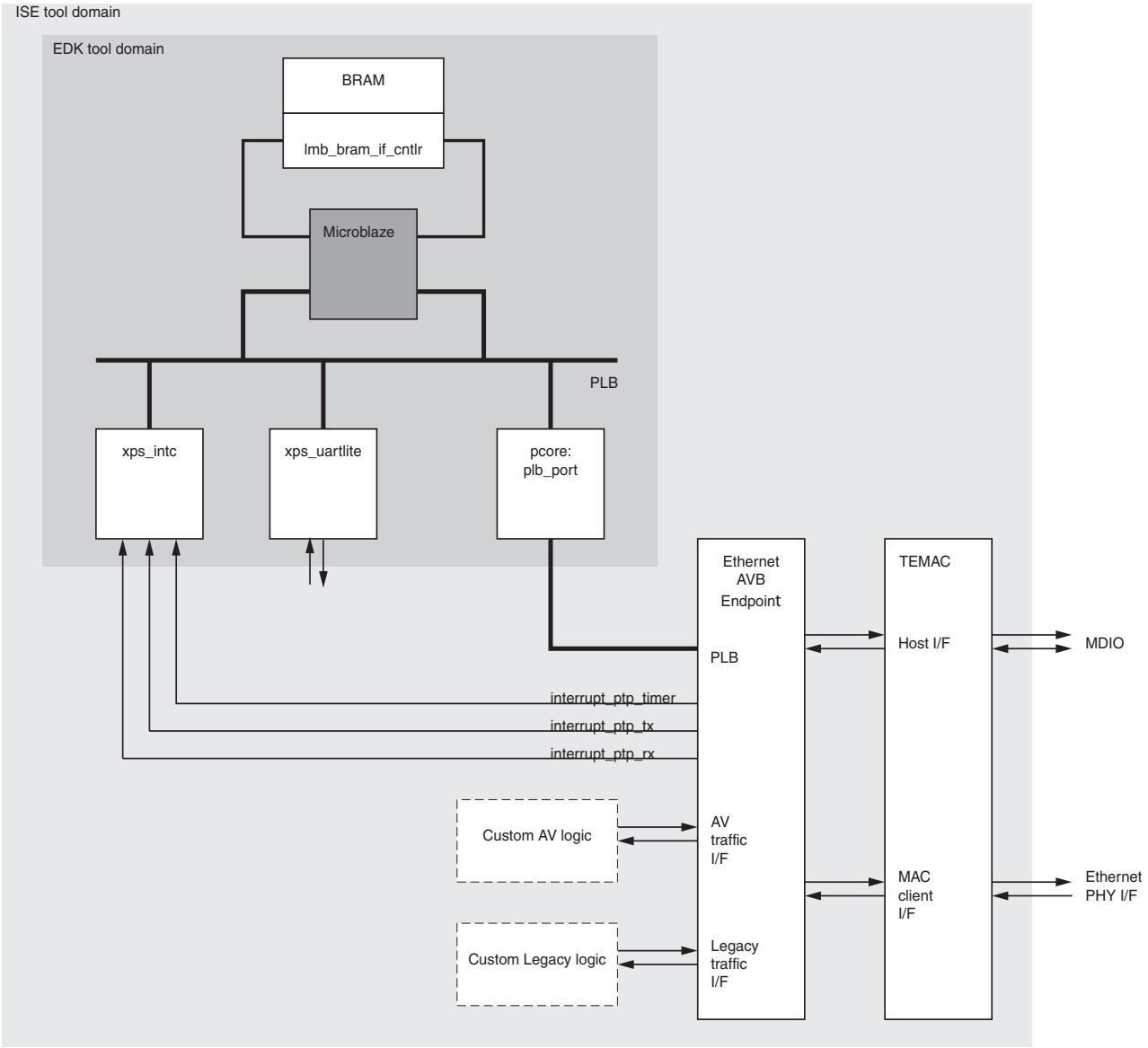

Figure 12-7: **Connection into an Embedded Processor Sub-system with an ISE Software Top-Level Project**

Figure 12-7 shows the implementation using an ISE® software top-level project. In this hierarchy, the embedded processor subsystem is created using an EDK project containing only the blocks illustrated in the *EDK tool domain* block. This EDK project is not the top level of the system and is instantiated as a black box subcomponent in a standard ISE software project as illustrated.

In this example:

- The EDK component is synthesized by the EDK tools; this block can then be left alone (unedited)
- All other components (for example, the *Custom AV logic*) can be created using a standard ISE software project. This flow should be familiar to a wider range of engineers than the EDK tool set.

The main advantages of this implementation hierarchy are in terms of possible faster development turn-around for synthesis/implementation run time. This is as a result that the EDK components will pre-exist in netlist format and do not have to be re-synthesized for each design iteration.

A final word of explanation is required for the EDK project illustrated in Figure 12-7. To assign the AVB software drivers running on the MicroBlaze processor to the Ethernet AVB Endpoint core, the *plb_port* pcore was created. This pcore is simply a bunch of wires to route through all of the PLB signals through to ports of the EDK block top level. In the Microprocessor Hardware Specification (.mhs) file, this pcore was assigned a base address matching that of the Ethernet AVB Endpoint "PLB Base Address" (in the generated netlist produced by the CORE Generator software). Then the AVB software drivers were assigned to the *plb_port* instance in the Microprocessor Software Specification (.mss) file.

Using the Xilinx XPS LocalLink Tri-Mode Ethernet MAC

The Ethernet AVB Endpoint core should be generated in the "EDK pcore Format" when connecting to the XPS LocalLink Tri-Mode Ethernet MAC core (xps_ll_temac).

Introduction

The xps_ll_temac is delivered with data path FIFO's (of configurable depth), optional TCP/IP Offload Engine (TOE) logic and various other optional features, all of which can be connected to Scatter Gather Direct Memory Access (DMA) Engines. Together with software drivers, this provides an entire ethernet networking stack (such as TCP/IP). All of this functionality is available in the EDK.

The xps_ll_temac uses a CORE Generator LogiCORE IP Ethernet MAs as a subcomponents. It is able to use either the soft core TEMAC product, or an Embedded Tri-Mode Ethernet MAC (available in certain Virtex devices). The xps_ll_temac functionality remains identical for either MAC implementation.

The integration of the Ethernet AVB Endpoint core with the xps_ll_temac extends the functionality of the xps_ll_temac to additionally include all of the features of AVB. Figure 5-2 provides an overview of this system: observe that the "AV Traffic Interface" remains available for custom logic to source and sink the time sensitive streaming data (e.g. audio / video data). Also refer to Figure 12-8 for a different perspective of this system.

xps_ll_temac configuration

To configure the xps_ll_temac with the required ports to interface with the Ethernet AVB Endpoint, ensure that the following option is set:

- C_TEMAC_AVB

Ensure, via xps_ll_temac software initialization that:

- Jumbo frames are disabled
- VLAN is enabled
- the MAC is set to operate in promiscuous mode (the TEMAC address filter is disabled).

For further information, please refer directly to the *xps_ll_temac product specification*.

System Overview: AVB capable xps_ll_temac

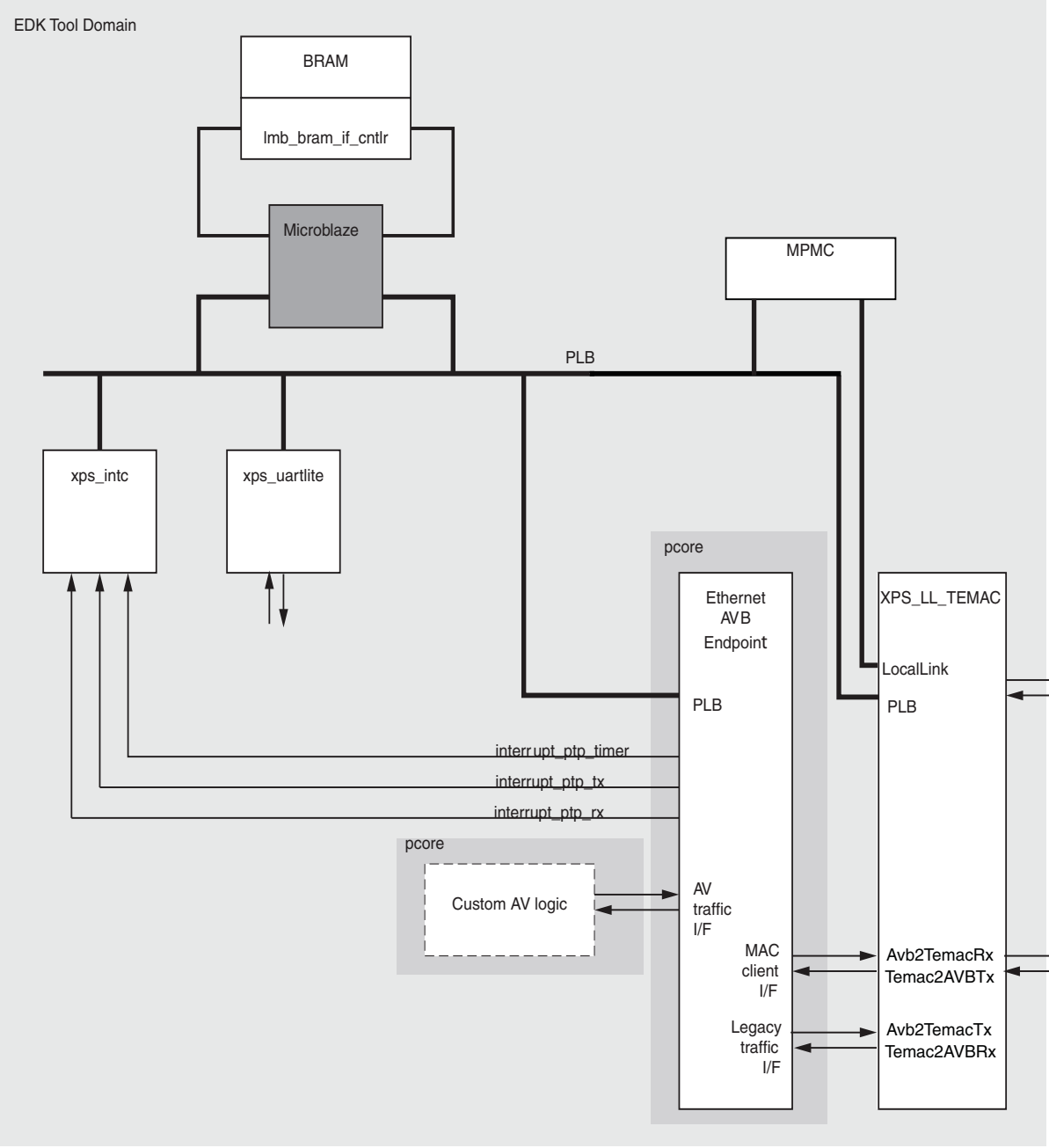

Figure 12-8: **Connection of the Ethernet AVB Endpoint Core into an Embedded Processor Sub-system**

Figure 12-8 illustrates the connection of the core to an embedded processor subsystem (MicroBlaze™ processor is illustrated). Observe that:

- The PLB can be shared across all peripherals as illustrated.
- The "Interrupt Signals" should be connected to the inputs of an interrupt controller module, for example, the *xps_intc* core provided with the EDK.
- The "Legacy Traffic Interface" and "Tri-Mode Ethernet MAC Client Interface" of the Ethernet AVB Endpoint core connect directly to the xps_ll_temac. This enables the xps_ll_temac to source and sink all legacy frame data. This legacy data is transferred to and from the processor domain over a LocalLink interface to a Multi-Port Memory Controller (MPMC); this contains scatter-gather DMA functionality and access to all processor memory such as external SDRAM (not illustrated). This allows software applications running on the processor to easily assemble and disassemble legacy ethernet packets.
- The "AV Traffic Interface" remains available for custom logic. This will be able to take priority over the processors legacy traffic as defined by the "P802.1Qav" component of the AVB specification.

Note that the embedded processor should be configured to use the software drivers provided with the core (see Chapter 13, "Software Drivers").

Ethernet AVB Endpoint Connections

Figure 12-8 illustrates the overall connections of the Ethernet AVB Endpoint core; only the "AV Traffic Interface" remains unconnected and is therefore available for custom logic.

All connections must be made in the EDK environment; please refer to Xilinx Platform Studio documentation. Extracts from a .mhs file will be included at the end of this section to further illustrate these connections.

Figure 12-9 illustrates the connections of the Ethernet AVB Endpoint core to the XPS LocalLink Tri-Mode Ethernet MAC (xps_ll_temac) core in detail. All connections, as shown, are logic-less connections. Observe that:

- The "Legacy Traffic Interface" and "Tri-Mode Ethernet MAC Client Interface" of the Ethernet AVB Endpoint core connect directly to the xps_ll_temac.
- The Ethernet transmitter client clock domain must always be connected to the `tx_clk` input of the Ethernet AVB Endpoint core. Additionally, the transmitter clock enable, as used with the TEMAC, must always be connected to the `tx_clk_en` input of the Ethernet AVB Endpoint core.
- The Ethernet receiver client clock domain must always be connected to the `rx_clk` input of the Ethernet AVB Endpoint core. Additionally, the receiver clock enable, as used with the TEMAC, must always be connected to the `rx_clk_en` input of the Ethernet AVB Endpoint core.

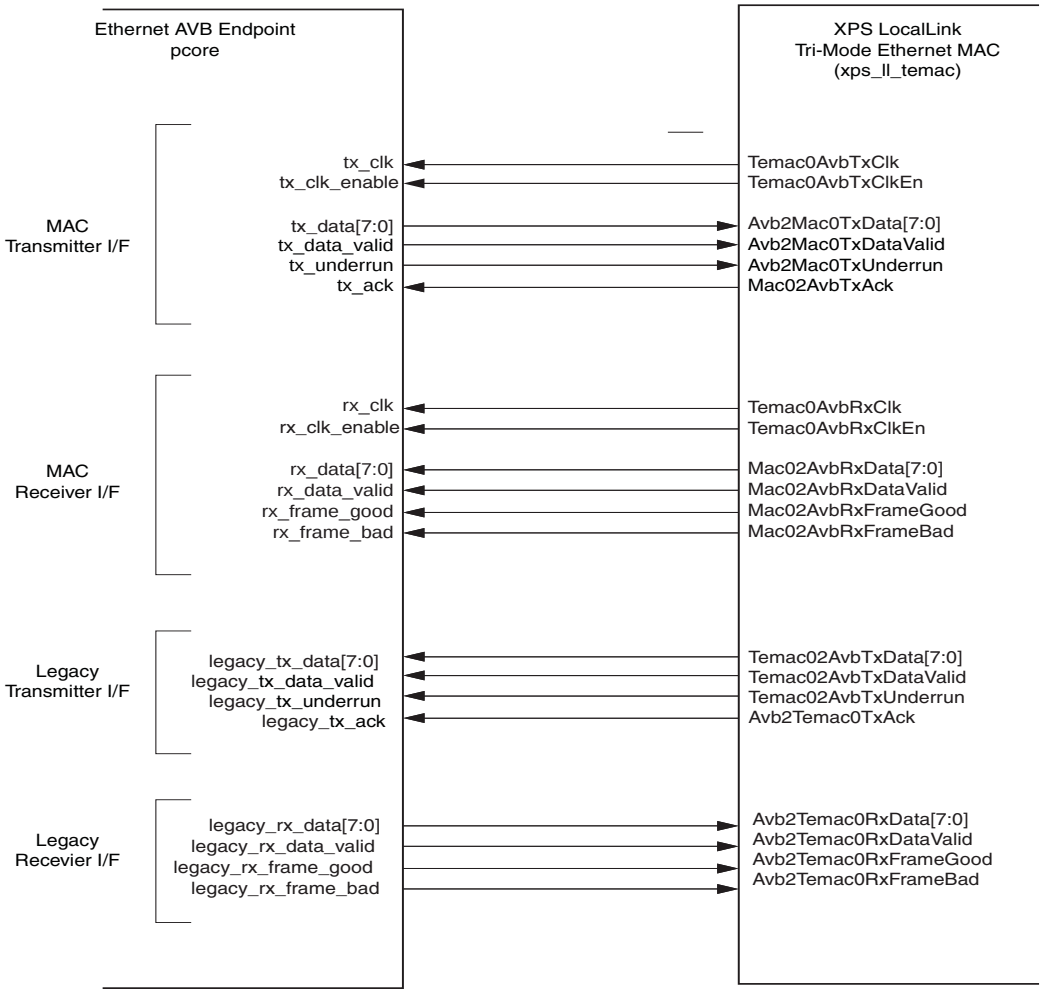

Figure 12-9: **Connection to the XPS LocalLink Tri-Mode Ethernet MAC**

MHS File Syntax

The following code extracts are taken from an XPS project which connected the Ethernet AVB Endpoint core to an instance of the xps_ll_temac. This design targeted the Virtex-5 family and implemented the xps_ll_temac using an Embedded Tri-Mode Ethernet MAC macro.

This MHS syntax is included for illustration/guideline purposes. It is recommended that the XPS GUI is used to import and connect pcore peripherals rather than by manually editing the .mhs file for a given project. Please refer to Xilinx Platform Studio documentation.

Certain lines are highlighted and commented to draw attention.

```
BEGIN xps_ll_temac
 PARAMETER INSTANCE = Hard_Ethernet_MAC
 PARAMETER C_NUM_IDELAYCTRL = 2
 PARAMETER C_IDELAYCTRL_LOC = IDELAYCTRL_X0Y4-IDELAYCTRL_X1Y5
 PARAMETER C_FAMILY = virtex5
 PARAMETER C_PHY_TYPE = 1
 PARAMETER C_TEMAC1_ENABLED = 0
 PARAMETER C_BUS2CORE_CLK_RATIO = 1
 PARAMETER C_TEMAC_TYPE = 0
 PARAMETER C_TEMAC0_PHYADDR = 0b00001
 PARAMETER HW_VER = 2.02.a
 PARAMETER C_TEMAC0_AVB = 1 # Enable AVB connections
 PARAMETER C_BASEADDR = 0x81c00000
 PARAMETER C_HIGHADDR = 0x81c0ffff
 BUS_INTERFACE SPLB = mb_plb
 BUS_INTERFACE LLINK0 = Hard_Ethernet_MAC_LLINK0
 PORT TemacIntc0_Irpt = Hard_Ethernet_MAC_TemacIntc0_Irpt
 PORT TemacPhy_RST_n = fpga_0_Hard_Ethernet_MAC_TemacPhy_RST_n_pin
 PORT GTX_CLK_0 = clk_125_0000MHzPLL0
 PORT REFCLK = clk_200_0000MHz
 PORT LlinkTemac0_CLK = clk_125_0000MHzPLL0
 PORT MII_TX_CLK_0 = fpga_0_Hard_Ethernet_MAC_MII_TX_CLK_0_pin
 PORT GMII_TXD_0 = fpga_0_Hard_Ethernet_MAC_GMII_TXD_0_pin
 PORT GMII_TX_EN_0 = fpga_0_Hard_Ethernet_MAC_GMII_TX_EN_0_pin
 PORT GMII_TX_ER_0 = fpga_0_Hard_Ethernet_MAC_GMII_TX_ER_0_pin
 PORT GMII_TX_CLK_0 = fpga_0_Hard_Ethernet_MAC_GMII_TX_CLK_0_pin
 PORT GMII_RXD_0 = fpga_0_Hard_Ethernet_MAC_GMII_RXD_0_pin
 PORT GMII_RX_DV_0 = fpga_0_Hard_Ethernet_MAC_GMII_RX_DV_0_pin
 PORT GMII_RX_ER_0 = fpga_0_Hard_Ethernet_MAC_GMII_RX_ER_0_pin
 PORT GMII_RX_CLK_0 = fpga_0_Hard_Ethernet_MAC_GMII_RX_CLK_0_pin
 PORT MDC_0 = fpga_0_Hard_Ethernet_MAC_MDC_0_pin
 PORT MDIO_0 = fpga_0_Hard_Ethernet_MAC_MDIO_0_pin
  # Connect as per Figure 12-9
 PORT Temac0AvbTxClk = Temac0AvbTxClk
 PORT Temac0AvbTxClkEn = Temac0AvbTxClkEn
 PORT Temac0AvbRxClk = Temac0AvbRxClk
 PORT Temac0AvbRxClkEn = Temac0AvbRxClkEn
 PORT Avb2Mac0TxData = Avb2Mac0TxData
 PORT Avb2Mac0TxDataValid = Avb2Mac0TxDataValid
 PORT Avb2Mac0TxUnderrun = Avb2Mac0TxUnderrun
 PORT Mac02AvbTxAck = Mac02AvbTxAck
 PORT Mac02AvbRxData = Mac02AvbRxData
 PORT Mac02AvbRxDataValid = Mac02AvbRxDataValid
 PORT Mac02AvbRxFrameGood = Mac02AvbRxFrameGood
 PORT Mac02AvbRxFrameBad = Mac02AvbRxFrameBad
 PORT Temac02AvbTxData = Temac02AvbTxData
 PORT Temac02AvbTxDataValid = Temac02AvbTxDataValid
 PORT Temac02AvbTxUnderrun = Temac02AvbTxUnderrun
 PORT Avb2Temac0TxAck = Avb2Temac0TxAck
 PORT Avb2Temac0RxData = Avb2Temac0RxData
 PORT Avb2Temac0RxDataValid = Avb2Temac0RxDataValid
 PORT Avb2Temac0RxFrameGood = Avb2Temac0RxFrameGood
 PORT Avb2Temac0RxFrameBad = Avb2Temac0RxFrameBad
END

BEGIN eth_avb_endpoint
 PARAMETER INSTANCE = eth_avb_endpoint_0
 PARAMETER HW_VER = 2.02.a
 PARAMETER C_MEM0_BASEADDR = 0xcc000000
```

```
 PARAMETER C_MEM0_HIGHADDR = 0xcc00ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT reset = sys_periph_reset
  # Connect as per Figure 12-9
 PORT tx_clk = Temac0AvbTxClk
 PORT tx_clk_en = Temac0AvbTxClkEn
 PORT rx_clk = Temac0AvbRxClk
 PORT rx_clk_en = Temac0AvbRxClkEn
 PORT tx_data = Avb2Mac0TxData
 PORT tx_data_valid = Avb2Mac0TxDataValid
 PORT tx_underrun = Avb2Mac0TxUnderrun
 PORT tx_ack = Mac02AvbTxAck
 PORT rx_data = Mac02AvbRxData
 PORT rx_data_valid = Mac02AvbRxDataValid
 PORT rx_frame_good = Mac02AvbRxFrameGood
 PORT rx_frame_bad = Mac02AvbRxFrameBad
  PORT legacy_tx_data = Temac02AvbTxData
 PORT legacy_tx_data_valid = Temac02AvbTxDataValid
 PORT legacy_tx_underrun = Temac02AvbTxUnderrun
 PORT legacy_tx_ack = Avb2Temac0TxAck
 PORT legacy_rx_data = Avb2Temac0RxData
 PORT legacy_rx_data_valid = Avb2Temac0RxDataValid
 PORT legacy_rx_frame_good = Avb2Temac0RxFrameGood
 PORT legacy_rx_frame_bad = Avb2Temac0RxFrameBad
 PORT rtc_clk = Temac0AvbTxClk
# Unused in this example: connect to custom pcores
  PORT av_tx_data = net_gnd
  PORT av_tx_valid = net_gnd
  PORT av_tx_done = net_gnd
# PORT av_tx_ack              =
# PORT av_rx_data             =
# PORT av_rx_valid            =
# PORT av_rx_frame_good       =
# PORT av_rx_frame_bad        =
# PORT rtc_nanosec_field      =
# PORT rtc_sec_field          =
# PORT clk8k                  =
# PORT rtc_nanosec_field_1722 =
# PORT interrupt_ptp_tx       =
# Connected Interrupts (to a xps_intc core)
  PORT interrupt_ptp_timer = AvbPtpInt
  PORT interrupt_ptp_rx = AvbRxInt
END
```

XILINX®

Chapter 13

Software Drivers

Software drivers delivered with the Ethernet AVB Endpoint core provide the following functions, which utilize the dedicated hardware within the core for the Precise Timing Protocol (PTP) *IEEE P802.1AS* specification:

- **Best Clock Master Algorithm (BMCA)** determines whether the core should operate in master clock or slave clock mode
- **PTP Clock Master** functions
- **PTP Clock Slave** functions that accurately synchronize the local Real Time Clock (RTC) to match that of the network clock master

The following definitions provide only a simplistic concept of PTP protocol operation. For detailed information about the PTP protocol, see the *IEEE P802.1AS* specification.

This chapter only describes the basic operation and some key components of the software drivers. The software driver code is documented such that the comments can be viewed by Doxygen and detailed descriptions of all aspects of the software are available throughout the code. This should allow customers to fully understand the operation of the provided software drivers and to edit the drivers for their own secret source applications.

Fundamentally, the slave "Real Time Clock" synchronization functions complete a software controlled phase-locked loop. Therefore, many implementations are possible. The provided software drivers implement a very simple software PLL implementation. However, this has been shown in hardware to provide excellent "Real Time Clock" synchronization results.

The document section "drivers/avb_v2_04_a/src" in Chapter 16 lists all of the C files delivered with the Ethernet AVB Endpoint core and provides a description of how the software is divided up between these files.

Clock Master

If the core is acting as clock master, the software drivers delivered with the core periodically sample the current value of the RTC and transmit this value to every device on the network using the *P802.1* defined *Sync* and *Follow-Up* PTP packets.

Clock Slave

If the core is acting as a clock slave, the local RTC is closely matched to the value and frequency of the network clock master. This is achieved, in part, by receiving the PTP *Sync* and *Follow-Up* frames transmitted across the network by the clock master (and containing the sampled RTC value of the master). The PTP mechanism also tracks the total routing delay across the network between the clock master and itself. The software drivers use this data, in conjunction with recent historical data, to calculate the error between its local RTC counter and that of the RTC clock master. The software then periodically calculates an RTC correction value and an updated increment rate, and these values are written to appropriate RTC configuration registers.

Because the drivers are provided as C code text files, they can be easily modified and designers can edit the files to provide their own secret source, or even to update the software drivers for *P802.1AS* specification changes.

Software System Integration

The software drivers for the Ethernet AVB Endpoint core need to be run on an embedded processor. In addition, they require instantiation into the overall software project, and then initialization.

An example software project file that performs the required steps is included with the core in the following location:

```
<component_name>/MyProcessorIPLib/drivers/
ethernet_avb_endpoint_v2_04_a/examples/xavb_example.c
```

This software example has been tested in a real system. For this reason, use this file for reference, along with the following descriptions:

- "Driver Instantiation"
- "Interrupt Service Routine Connections"
- "Core Initialization"
- "Ethernet AVB Endpoint Setup"
- "Starting and Stopping the AVB Drivers"

Note: Unless you are already familiar with the Xilinx Embedded Development Kit (EDK), see the EDK documentation to follow the steps described.

Driver Instantiation

Software driver instantiation for the Ethernet AVB Endpoint core follows the standard EDK model used for all EDK IP cores and as recommended for all user defined pcores (see the EDK documentation). Initialization of the driver requires that an instance of the driver is instantiated, assigned a base address within the PLB address range, and configured using the standardized cores `CfgInitialize` function.

For example, in the user software, the AVB drivers can be instanced as follows:

```
/* Allocate an instance of the XAvb device driver */
static XAvb Avb;
int Status;
XAvb_Config *AvbConfigPtr;
.
/* Initialize AVB Driver */
AvbConfigPtr = XAvb_LookupConfig(AVB_DEVICE_ID);
Status = XAvb_CfgInitialize(&Avb,
                     AvbConfigPtr,
                     AvbConfigPtr->BaseAddress);
```

In the previous example, the `AVB_DEVICE_ID` is defined in the `xparameters.h` file, automatically generated by the EDK tools as a result of the software driver instance and the hardware instance of the Ethernet AVB Endpoint core in the Microprocessor Software Specification (.mss) file and the Microprocessor Hardware Specification (.mhs) files.

When the core has been generated in the Standard CORE Generator™ format (see "Core Delivery Format"), the value of the base address used for the hardware instance in the .mhs file must match the value of the PLB base address which was selected during the Ethernet AVB Endpoint core generation.

When the core has been generated in the EDK pcore format, the value of the PLB base address will be automatically configured by XPS.

Interrupt Service Routine Connections

The Ethernet AVB Endpoint core creates three interrupt output signals: `interrupt_ptp_timer`, `interrupt_ptp_tx` and `interrupt_ptp_rx`. It is recommended that these be connected to the interrupt input ports of a `xps_intc` core: this is a standard interrupt controller core, complete with associated software drivers, which are available with the EDK.

In this version of the Ethernet AVB Endpoint core, only the `interrupt_ptp_timer` and `interrupt_ptp_rx` interrupts are required by the software drivers. The functionality provided by the `interrupt_ptp_tx` interrupt signal, as used in previous software driver versions, has been replaced with polling functionality to reduce the overall interrupt driver overhead.

The two hardware interrupt signals required need to be connected to the following interrupt routine service functions:

- `interrupt_ptp_timer` needs to call the function XAvb_PtpTimerInterruptHandler()
- `interrupt_ptp_rx` needs to call the function XAvb_PtpRxInterruptHandler()

Again, see the provided software example file that performs these steps.

Core Initialization

When Using a LogiCORE IP Tri-Mode Ethernet MAC

Note: When connecting to the XPS LocalLink Tri-Mode Ethernet MAC (xps_ll_temac), available in EDK, the MAC is delivered with its own drivers and the functionality of this subsection is not required.

The Xilinx LogiCORE™ IP "Tri-Mode Ethernet MACs" require initialization of the MDIO clock frequency (the MDC signal) and requires specific non-default configuration (VLAN enabled, Flow Control disabled). The following lines of code of perform these steps.

```
/* Configure MDIO Master in TEMAC - MUST be done before any MDIO
accesses */
  XAvbMac_WriteConfig(InstancePtr->Config.BaseAddress,
                      XAVB_MAC_MGMT_REG_OFFSET,
                      0x0000004F);

  /* Disable TEMAC Flow Control */
  XAvbMac_WriteConfig(InstancePtr->Config.BaseAddress,
                      XAVB_MAC_FC_REG_OFFSET,
                      0x0);

  /* Initialize TEMAC by enabling Tx and Rx with VLAN capability */
  XAvbMac_WriteConfig(InstancePtr->Config.BaseAddress,
                     XAVB_MAC_TX_REG_OFFSET,
                     (XAVB_MAC_TX_ENABLE_MASK |
XAVB_MAC_TX_VLAN_ENABLE_MASK));
```

Ethernet AVB Endpoint Setup

This section describes the main elements that you may have to modify in order to operate the Ethernet AVB Endpoint software in their application.

System-Specific Defines in xavb_hw.h

This header file assumes that the rtc_clk input is connected to a 125 MHz clock source. If a clock with a different frequency is connected to this input, then the following #define should be edited so that the increment written to the "RTC Increment Value Control Register" matches the RTC clock.

```
#define XAVB_RTC_INCREMENT_NOMINAL_RATE 0x00800000
```

System-Specific Defines in xavb.h

The timestamp reference plane is defined by *IEEE P802.1AS* to be at the PHY and since the Ethernet AVB Endpoint captures the timestamp when the first symbol following the SFD is seen at the Ethernet MAC Client interface, the software needs to know the fixed latency values through the MAC and PHY. The following two #defines should be edited to store the values (in nanoseconds) of the ingress and egress delays through the PHY that is being used in the system. The values are set to 0 by default.

```
#define XAVB_TX_MAC2PHY_LATENCY_IN_NS 0
#define XAVB_RX_PHY2MAC_LATENCY_IN_NS 0
```

You should also update the following #define if there is a known asymmetry in the propagation delay on the link. This #define models the per-port global variable "delayAsymmetry" as defined in *IEEE P802.1AS* and should be edited based on the description given in this specification. However, the current implementation uses truncated nanoseconds rather than the scaled Ns type. The value is set to 0 by default.

```
#define XAVB_PROP_DELAY_ASYMMETRY 0
```

Setting up SourcePortIdentity (and Default TX PTP Messages)

The TX Packet buffers are pre-initialized with default values for all of the possible fields in each message. However, in order for the Ethernet AVB Endpoint software drivers to run correctly the following fields need to be written with sensible values.

- sourcePortIdentity in all TX PTP default messages
- grandmasterIdentity in TX PTP Announce message
- pathSequence (ClockIdentity[1]) in TX PTP Announce message

The example design `xavb_example.c` provides a simple mechanism to achieve this using the following #defines.

```
#define ETH_SOURCE_ADDRESS_EUI48_HIGH  0xFFEEDD
#define ETH_SOURCE_ADDRESS_EUI48_LOW   0xCCBBAA
```

You can edit the #defines above to be the Ethernet Source Address for the device and the example software then provides code that translates this address into an XAvb_PortIdentity struct. The function XAvb_SetupSourcePortIdentity() is called with the XAvb_PortIdentity struct and writes it to the appropriate fields in the TX PTP Buffer. Additionally it stores it in the XAvb struct as the following member:

```
/** Contains the local port Identity information */
XAvb_PortIdentity portIdLocal;
```

The example software also provides an example of how to write the Ethernet Source Address into all TX PTP packet buffers.

Setting up GrandMaster Discontinuity Callback Handler

The Ethernet AVB Endpoint software defines a callback routine which is called when the endpoint switches between being a Master and a Slave (or vice versa), or when it loses PTP lock. The application software must define a callback handler for this otherwise an error will be asserted. The example software provides an example of this as follows:

```
/** Function Prototype */
static void GMDiscontinuityHandler(void *CallBackRef,
                           unsigned int TimestampsUncertain);

/** Main function in this example */
main() {
/** ... */
  XAvb_Config *AvbConfigPtr;

/** Setup the handler that will be called if the PTP drivers
   * identify a possible discontinuity in GrandMaster time. */
  XAvb_SetGMDiscontinuityHandler(&Avb, GMDiscontinuityHandler, &Avb);

/** ... */
/*****************************************************************/
/**
```

```
* This function is the handler which will be called if the PTP drivers
* identify a possible discontinuity in GrandMaster time.
* This handler provides an example of how to handle this situation -
* but this function is application specific.
*
* @param CallBackRef contains a callback reference from the driver, in
* this case it is the instance pointer for the AVB driver.
* @param TimestampsUncertain - a value of 1 indicates that there is a
* possible discontinuity in GrandMaster time. A value of 0
* indicates that Timestamps are no longer uncertain.
*****************************************************************/
static void GMDiscontinuityHandler(void *CallBackRef,
                            unsigned int TimestampsUncertain)
{
xil_printf("\r\nGMDiscontinuityHandler: Timestamps are now %s\r\n",
          TimestampsUncertain ? "uncertain" : "certain");
}
```

Starting and Stopping the AVB Drivers

The default state after driver initialization is for the AVB drivers to be inactive. After the Ethernet link has been established, the drivers can be started using the following function call. This will begin operation of the IEEE802.1 AS PTP protocol.

```
XAvb_Start(InstancePtr);
```

Before starting the drivers, ensure that the Ethernet PHY has successfully auto-negotiated a full duplex link at either 100 Mbps or 1 Gbps Ethernet speeds. Early implementations may also require the completion of an LLDP (Link Layer Discovery Protocol) function. LLDP has been used in early AVB implementations to negotiate support of AVB between peer devices for interoperability (AVB standards are still considering the use of LLDP). LLDP is not currently included in our software drivers or example file.

The AVB drivers can be stopped at any time (to halt the IEEE802.1 AS PTP protocol) by calling the following function:

```
XAvb_Stop(InstancePtr);
```

The software example included will halt the drivers whenever the Ethernet PHY Auto-Negotiation indicates that it has lost the link, or has negotiation to an unsupported ethernet mode (for example, half duplex).

XILINX®

Chapter 14

Quick Start Example Design

The quick start steps provided in this chapter let you quickly generate an Ethernet AVB Endpoint core, run the design through implementation with the Xilinx tools, and simulate the design using the provided demonstration test bench. For detailed information about the Standard CORE Generator example design, see Chapter 15, "Detailed Example Design (Standard Format)." For detailed information about the EDK pcore example design, seeChapter 16, "Detailed Example Design (EDK format)."

Overview

The Ethernet AVB Endpoint example design consists of the following:

- Ethernet AVB Endpoint core netlist
- Example design HDL top-level and associated HDL files
- Demonstration test bench to exercise the example design

The Ethernet AVB Endpoint example design has been tested using Xilinx® ISE® software v12.2, Cadence Incisive Enterprise Simulator (IES) v9.2, Mentor Graphics ModelSim v 6.5c, and Synopsys VCS and VCS MX 2009.12.

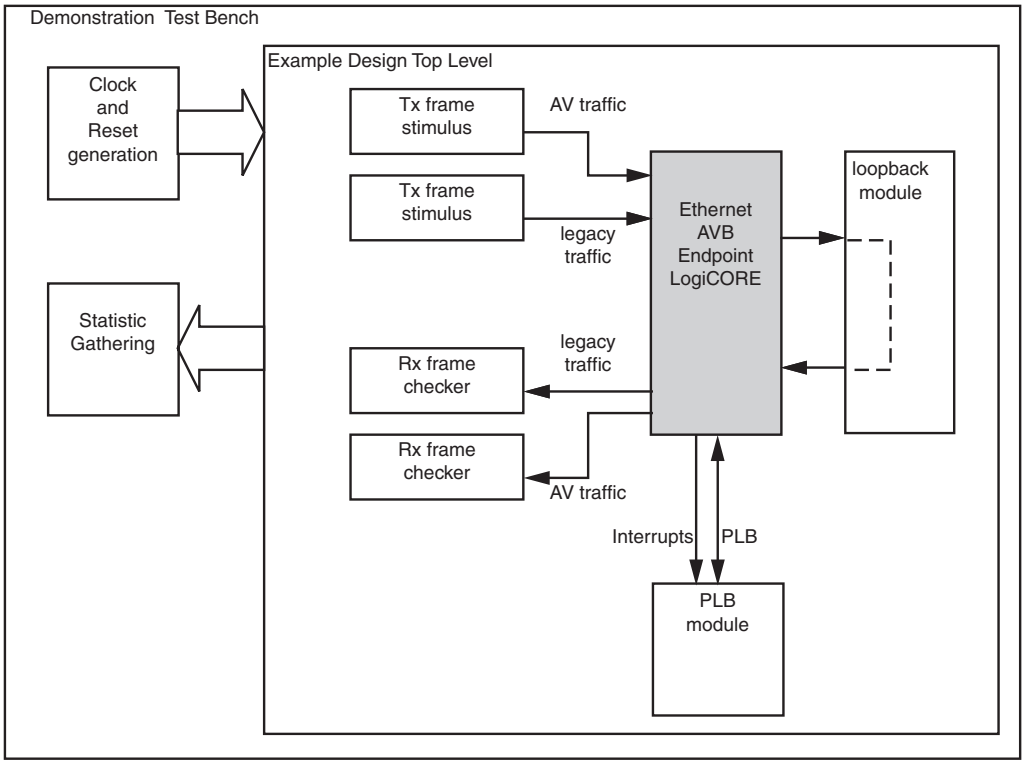

Figure 14-1: **Ethernet AVB Endpoint Example Design and Test Bench**

Generating the Core

This section provides detailed instructions for generating the Ethernet AVB Endpoint example design core.

To generate the core:

1. Start the CORE Generator™ tool.

   For general help with starting and using CORE Generator software on your system, see the documentation supplied with the ISE software, including the *CORE Generator Guide*. These documents can be downloaded from: www.xilinx.com/support/software_manuals.htm

2. Create a new project.
3. For project options, select the following:
   - A Virtex®-6, Virtex-5, Spartan®-3, Spartan-3E, Spartan-3A/3A DSP or Spartan-6 device to generate the default Ethernet AVB Endpoint core.
   - In the Design Entry section, select VHDL or Verilog; then select Other for Vendor.
4. Locate the Ethernet AVB Endpoint core in the taxonomy tree, listed under one of the following:
   - Automotive & Industrial/Automotive
   - Communications & Networking/Ethernet
   - Communications & Networking/Networking
   - Communications & Networking/Telecommunications
5. Double-click the core name. A message may appear to indicate the limitations of the Simulation Only Evaluation license.
6. Click OK; the core customization screen appears.

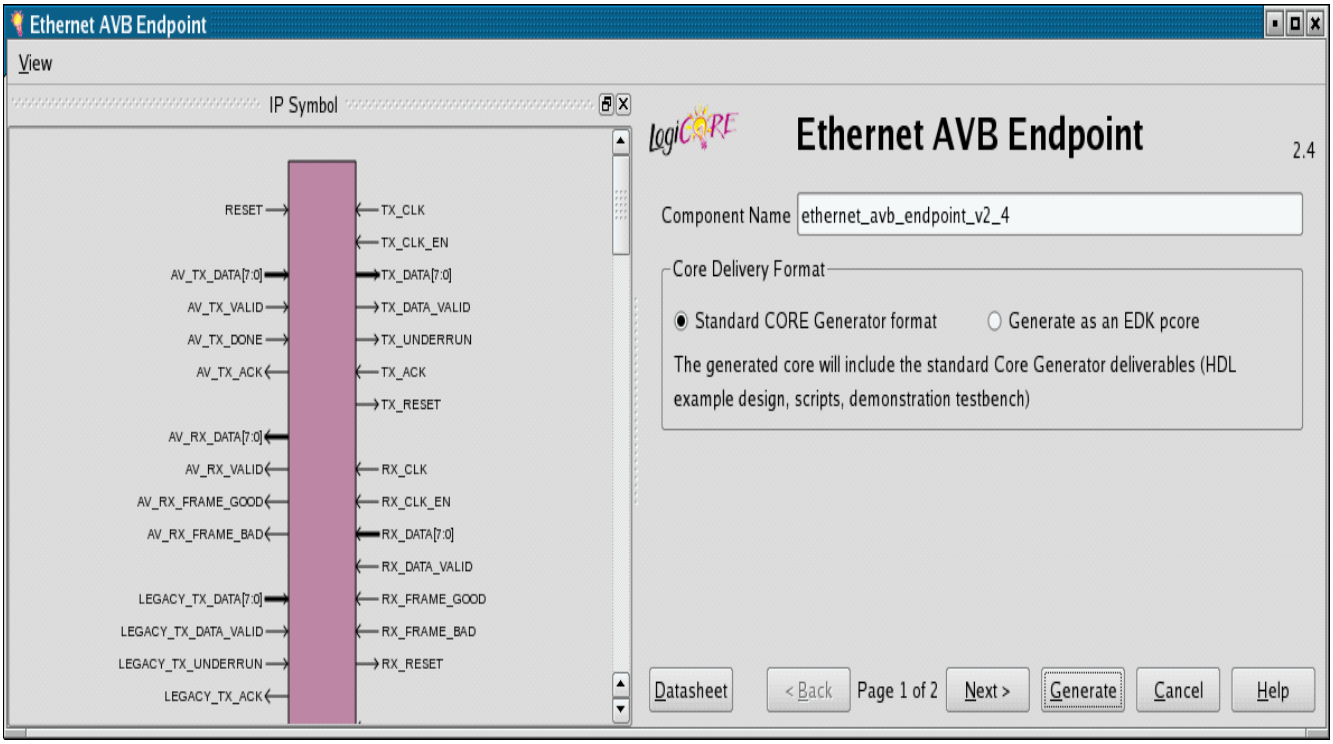

Figure 14-2: **Ethernet AVB Endpoint Core Customization Screen**

7. Enter a core instance name in the Component Name field.
8. Maintain the default options on GUI page 1 so that *Standard CORE Generator format* is selected.
9. Click Generate to deliver the core using the default options.

The default core and its supporting files, including the example design, are generated in your project directly. For a detailed description of the design example files and directories, see Chapter 15, "Detailed Example Design (Standard Format)."

Implementing the Example Design

After the core is generated, the netlists and example design can be processed by the Xilinx implementation tools. The generated output files include several scripts to assist you in running the Xilinx software.

To implement the Ethernet AVB Endpoint example design core:

From the CORE Generator software project directory window, type the following:

Linux

```
% cd <project_dir>/<component_name>/implement
% ./implement.sh
```

Windows

```
> cd <project_dir>\<component_name>\implement
> implement.bat
```

These commands execute a script that synthesizes, builds, maps, and place-and-routes the example design. The script then creates gate-level netlist HDL files in either VHDL or Verilog, along with associated timing information (SDF) files.

Simulating the Example Design

Setting up for Simulation

To run functional and timing simulations you must have the Xilinx Simulation Libraries compiled for your system. See the Compiling Xilinx Simulation Libraries (COMPXLIB) in the *Xilinx ISE Synthesis and Verification Design Guide*, and the *Xilinx ISE Software Manuals and Help*. You can download these documents from:
www.xilinx.com/support/software_manuals.htm

Functional Simulation

This section provides instructions for running a functional simulation of the Ethernet AVB Endpoint core using either VHDL or Verilog. The functional simulation model is provided when the core generated; implementing the core before simulation is not required.

To run a VHDL or Verilog functional simulation of the example design:

1. Open a command prompt or shell, then set the current directory to:
   `<project_dir>/<component_name>/simulation/functional/`
2. Launch the simulation script:

   ```
   ModelSim: vsim -do simulate_mti.do
   IES: ./simulate_ncsim.sh
   VCS: ./simulate_vcs.sh (Verilog only)
   ```

The simulation script compiles the functional simulation model, the example design files, the demonstration test bench, and adds relevant signals to a wave window. It then runs the simulation to completion. After completion, you can inspect the simulation transcript and waveform to observe the operation of the core.

Timing Simulation

This section contains instructions for running a timing simulation of the Ethernet AVB Endpoint core using either VHDL or Verilog. A timing simulation model is generated when run through the Xilinx tools using the implementation script. You must implement the core before attempting to run timing simulation.

To run a VHDL or Verilog timing simulation of the example design:

1. Run the implementation script (see "Implementing the Example Design," page 141).
2. Open a command prompt or shell, then set the current directory to:
   `<project_dir>/<component_name>/simulation/timing/`
3. Launch the simulation script:

   ModelSim: **`vsim -do simulate_mti.do`**

   IES: **`./simulate_ncsim.sh`**

   VCS: **`./simulate_vcs.sh (Verilog only)`**

The simulator script compiles the gate-level model and the demonstration test bench, adds relevant signals to a wave window, and then runs the simulation to completion. You can then inspect the simulation transcript and waveform to observe the operation of the core.

What's Next?

The Ethernet AVB Endpoint core can be delivered in two different formats, selectable from page 1 the CORE Generator Customization GUI:

- Standard CORE Generator software format (provided for the standard ISE software environment)

  For detailed information about the core delivery using the Standard CORE Generator software format, including example design information, guidelines for modifying the design and extending the test bench, see Chapter 15, "Detailed Example Design (Standard Format).".

- Generate as an EDK pcore (provided for the Embedded Development Kit)

  For detailed information about the core delivery for the Embedded Developments Kit (EDK), see Chapter 16, "Detailed Example Design (EDK format)."

XILINX®

Chapter 15

Detailed Example Design (Standard Format)

This chapter provides detailed information about the core when generated for the Standard CORE Generator™ software format. This option is selected from page 1 of the customization GUI and will deliver the core with the standard CORE Generator software directory structure (used by many LogiCORE™ IP systems including all other CORE Generator software Ethernet cores).

This chapter provides detailed information on the core and example design, including a description of files and the directory structure generated by the Xilinx CORE Generator software, the purpose and contents of the provided scripts, the contents of the example HDL wrappers, and the operation of the demonstration test bench.

Please refer to Chapter 16, "Detailed Example Design (EDK format)." when targeting the Embedded Development Kit.

- **<project directory>**
  Top-level project directory; name is user-defined.
  - <project directory>/<component name>
    Core release notes file
    - <component name>/doc
      Product documentation
    - <component name>/example design
      Verilog or VHDL design files
    - <component name>/implement
      Implementation script files
      - implement/results
        Results directory, created after implementation scripts are run, and contains implement script results
    - <component name>/simulation
      Simulation scripts
      - simulation/functional
        Functional simulation files
      - simulation/timing
        Timing simulation files

- <component_name>/drivers/v2_04_a
  Files for compiling the low-level drivers provided with the core
  - drivers/avb_v2_04_a/data
    Data files for automatic integration into Xilinx Platform Studio
  - drivers/avb_v2_04_a/examples
    An application example using the low-level driver files
  - drivers/avb_v2_04_a/src
    Low-level driver source C files

Directory and File Contents

The core directories and their associated files are defined in the following tables.

<project directory>

The project directory contains all the CORE Generator software project files.

Table 15-1: **Project Directory**

Name	Description
<project_dir>	
<component_name>.ngc	Top-level netlist. This is instantiated by the Verilog or VHDL example design.
<component_name>.v[hd]	Verilog or VHDL simulation model; UniSim-based.
<component_name>.v{ho|eo}	Verilog or VHDL instantiation template for the core.
<component_name>.xco	Log file that records the settings used to generate a core. An XCO file is generated by the CORE Generator software for each core that it creates in the current project directory. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	List of files delivered with the core.

Back to Top

<project directory>/<component name>

The <component name> directory contains the release notes file provided with the core, which may include last-minute changes and updates.

Table 15-2: **Component Name Directory**

Name	Description
<project_dir>/<component_name>	
eth_avb_endpoint_readme.txt	Core release notes file

Back to Top

<component name>/doc

The doc directory contains the PDF documentation provided with the core.

Table 15-3: **Doc Directory**

Name	Description
<project_dir>/<component_name>/doc	
eth_avb_endpoint_ds677.pdf	Ethernet AVB Endpoint Data Sheet
eth_avb_endpoint_ug492.pdf	Ethernet AVB Endpoint User Guide

Back to Top

<component name>/example design

The example design directory contains the example design files provided with the core. For more information, see "Example Design," page 152.

Table 15-4: **Example Design Directory**

Name	Description
<project_dir>/<component_name>/example_design/	
<component_name>_example_design.ucf	Example User Constraints File (UCF) provided for the example design.
<component_name>_example_design.v[hd]	Top-level file that allows the example design to be implemented in a device as a standalone design.
tx_frame_stimulus.v[hd]	An HDL file which is capable of producing Ethernet frames at maximum line-rate and containing a predictable pattern in the data field.
temac_loopback_shim.v[hd]	An HDL file which sits in the place of an Ethernet MAC (an Ethernet MAC is required in a real system). This file loops back the data from the transmitter client to the receiver client.

Table 15-4: **Example Design Directory *(Cont'd)***

Name	Description
rx_frame_checker.v[hd]	An HDL file which is capable of receiving Ethernet frames at maximum line rate. This will check the data contained in each Ethernet frame received against a predictable pattern. This file partners the tx_frame_stimulus file.
plb_client_logic.v[hd]	An HDL file that sits in the place of an embedded microprocessor (an embedded microprocessor is required in a real system), which provides stimulus to the PLB, performing write and reads that initiate PTP frame transmission.

Back to Top

<component name>/implement

The implement directory contains the core implementation script files.

Table 15-5: **Implement Directory**

Name	Description
<project_dir>/<component_name>/implement	
implement.sh	LINUX shell script that processes the example design through the Xilinx tool flow. See "Implementation Scripts," page 151 for more information.
implement.bat	Windows batch file that processes the example design through the Xilinx tool flow. See "Implementation Scripts," page 151 for more information.
xst.prj	XST project file for the example design (VHDL only); it enumerates all of the VHDL files that need to be synthesized.
xst.scr	XST script file for the example design.

Back to Top

implement/results

The results directory is created by the implement script, after which the implement script results are placed in the results directory.

Table 15-6: **Results Directory**

Name	Description
<project_dir>/<component_name>/implement/results	
routed.v[hd]	Back-annotated SimPrim-based model used for timing simulation.
routed.sdf	Timing information for simulation.

Back to Top

<component name>/simulation

The simulation directory and subdirectories that provide the files necessary to test a Verilog or VHDL implementation of the example design. For more information, see "Example Design," page 152.

Table 15-7: **Simulation Directory**

Name	Description
<project_dir>/<component_name>/simulation	
demo_tb.v[hd]	The demonstration test bench for the example design. Instantiates the example design (the Device Under Test (DUT)), generates clocks, resets, and gathers statistics as the simulation is run.

Back to Top

simulation/functional

The functional directory contains functional simulation scripts provided with the core.

Table 15-8: **Functional Directory**

Name	Description
<project_dir>/<component_name>/simulation/functional	
simulate_mti.do	ModelSim macro file that compiles Verilog or VHDL sources and runs the functional simulation to completion.
wave_mti.do	ModelSim macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_mti.do` macro file.
simulate_ncsim.sh	IES script file that compiles the Verilog or VHDL sources and runs the functional simulation to completion.

Table 15-8: **Functional Directory *(Cont'd)***

Name	Description
wave_ncsim.sv	IES macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_ncsim.sh` script file.
simulate_vcs.sh	VCS script file that compiles the Verilog sources and runs the functional simulation to completion.
vcs_commands.key	This file is sourced by VCS at the start of simulation; it configures the simulator.
vcs_session.tcl	VCS macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_vcs.sh` script file.

Back to Top

simulation/timing

The timing directory contains timing simulation scripts provided with the core.

Table 15-9: **Timing Directory**

Name	Description
<project_dir>/<component_name>/simulation/timing	
simulate_mti.do	ModelSim macro file that compiles Verilog or VHDL sources and runs the timing simulation to completion.
wave_mti.do	ModelSim macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_mti.do` macro file.
simulate_ncsim.sh	IES script file that compiles the Verilog or VHDL sources and runs the timing simulation to completion.
wave_ncsim.sv	IES macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_ncsim.sh` script file.
simulate_vcs.sh	VCS script file that compiles the Verilog sources and runs the timing simulation to completion.
vcs_commands.key	File sourced by VCS at the start of simulation; it configures the simulator.
vcs_session.tcl	VCS macro file that opens a wave window and adds signals of interest to it. It is called by the `simulate_vcs.sh` script file.

Back to Top

<component_name>/drivers/v2_04_a

A directory containing the software device drivers for the Ethernet AVB Endpoint core and associated supporting files.

drivers/avb_v2_04_a/data

The driver data directory contains the data files for automatic generation of parameter specific files when integrated into Platform Studio.

Table 15-10: **Driver Data Directory**

Name	Description
<project_dir>/<component_name>/drivers/avb_v2_04/data	
avb_v2_1_0.mdd	Current MDD file used, including the version of the tools interface.
avb_v2_1_0.tcl	Used to provide design rule checks within Xilinx Platform Studio.

Back to Top

drivers/avb_v2_04_a/examples

The driver examples directory contains an application example using the low-level driver files.

Table 15-11: **Driver Example Directory**

Name	Description
<project_dir>/<component_name>/drivers/avb_v2_04/examples	
xavb_example.c	Contains a very basic example design of using the AVB driver.

Back to Top

drivers/avb_v2_04_a/src

The driver source (src) directory contains the low-level driver source C files.

Table 15-12: **Driver Source Directory**

Name	Description
<project_dir>/<component_name>/drivers/ avb_v2_04/src	
Makefile	Makefile to compile the drivers; used by Platform studio.
xavb.h	Main header file for the XAvb driver. The file provides the constants, type definitions and function templates which are required to initialize and run the IEEE802.1AS Precise Timing Protocol (PTP). This defines the level 1 device driver for the Ethernet AVB Endpoint core.
xavb_g.c	Contains a configuration structure that holds all the configuration values required, per single instance, of the device driver.
xavb.c	Provides the top-level function calls for the Ethernet AVB Endpoint level 1 device driver.
xavb_ptp_packets.c	Provides the functions which are required for the creation of PTP frames for transmission and for the decode of received PTP frames.
xavb_ptp_bmca.c	Provides the functions which are required for the PTP Best Master Clock Algorithm (BMCA).
xavb_rtc_sync.c	Provides the functions which are required to synchronize the local version of the Real Time Counter (RTC), when operating as a slave, to that of the network clock master.
xavb_hw.h	Contains all the constant definitions and the bare minimum of functions / function templates which are required for register read/write access. This defines the low level 0 device driver for the Ethernet AVB Endpoint core.
xavb_hw.c	This file partners the xavb_hw.h header file and implements the functions for which avb_hw.h contained a template.

Back to Top

Implementation Scripts

The implementation script is either a shell script or batch file that processes the example design through the Xilinx tool flow and is one of the following locations:

Linux

```
<project_dir>/<component_name>/implement/implement.sh
```

Windows

```
<project_dir>/<component_name>/implement/implement.bat
```

The implement script performs the following steps:

1. HDL example design files are synthesized using XST.
2. Ngdbuild is run to consolidate the core netlist and the example design netlist into the NGD file containing the entire design.
3. Design is mapped to the target technology.
4. Design is placed-and-routed on the target device.
5. Static timing analysis is performed on the routed design using `trce`.
6. A bitstream is generated.
7. Netgen runs on the routed design to generate a VHDL or Verilog netlist (as appropriate for the Design Entry project setting) and timing information in the form of SDF files.

The Xilinx tool flow generates several output and report files that are saved in the following directory (which is created by the implement script):

```
<project_dir>/<component_name>/implement/results
```

Simulation Scripts

Functional Simulation

The test script is a ModelSim, IES, or VCS macro that automates the simulation of the test bench and is in the following location:

```
<project_dir>/<component_name>/simulation/functional/
```

The test script performs the following tasks:

- Compiles the structural UniSim simulation model
- Compiles HDL example design source code
- Compiles the demonstration test bench
- Starts a simulation of the test bench
- Opens a Wave window and adds signals of interest
- Runs the simulation to completion

Timing Simulation

The test script is a ModelSim, IES, or VCS macro that automates the simulation of the test bench and is in the following location:

```
<project_dir>/<component_name>/simulation/timing/
```

The test script performs the following tasks:

- Compiles the SimPrim-based gate level netlist simulation model
- Compiles the demonstration test bench
- Starts a simulation of the test bench using back-annotated timing information (SDF)
- Opens a Wave window and adds signals of interest
- Runs the simulation to completion

Example Design

Figure 15-1 illustrates the complete example design for the Ethernet AVB Endpoint. Individual sub-blocks are described in the following sections.

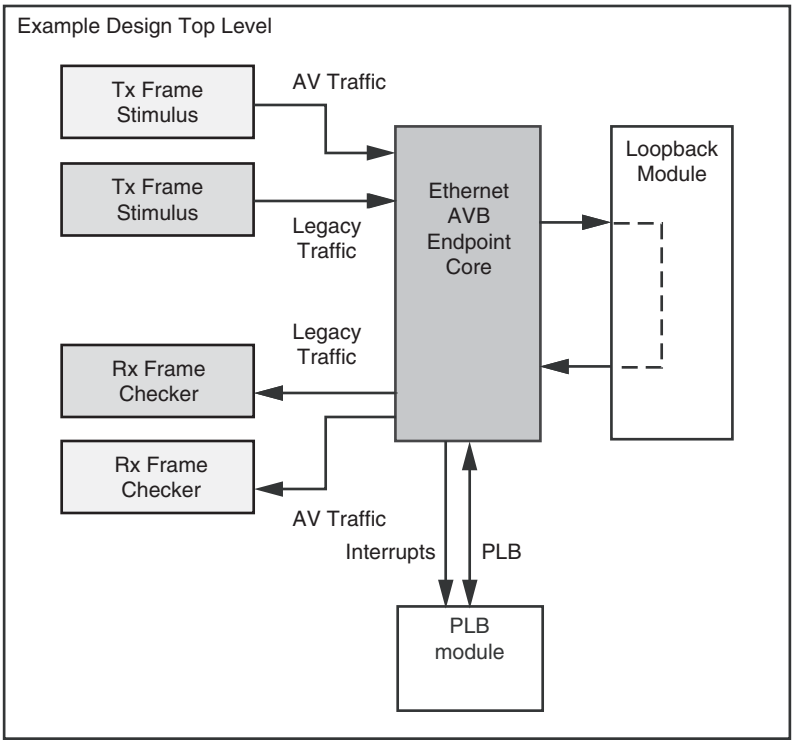

Figure 15-1: **Example Design HDL for the Ethernet AVB Endpoint**

Note: The example design is designed to allow the core, in isolation, to be tested and to demonstrate some of the functionality of the core, and does not create a realistic implementation. In a real system the loopback module should be replaced with an Ethernet MAC, the PLB module should be replaced with an embedded processor, and the frame stimulus and checker modules should be replaced with the desired AV and Legacy client functionality.

Top-Level Example Design HDL

The following files describe the top-level example design for the Ethernet AVB Endpoint core.

VHDL

<project_dir>/*<component_name>*/example_design/*<component_name>_example_design.vhd*

Verilog

<project_dir>/*<component_name>*/example_design/*<component_name>_example_design.v*

The example design HDL top level contains the following:

- An instance of the Ethernet AVB Endpoint core
- Two instances of an Ethernet Frame Stimulus block, configured differently and connected as follows:
  - One instance is connected to the AV transmitter interface, configured to produce VLAN Ethernet frames with a priority of 3.
  - A second instance is connected to the Legacy transmitter interface, configured to produce standard Ethernet frames without a VLAN field
- An instance of a Loopback Module, instantiated in place of where an Ethernet MAC should exist, enables the example design to be standalone. All AV and Legacy frames transmitted are then looped back and received at the corresponding AV and Legacy receive client interfaces.
- Two instances of an Ethernet Frame Checker block, configured differently and connected as follows:
  - One instance is connected to the AV receiver interface, configured to expect the VLAN frames produced by the AV Frame Stimulus block
  - A second instance is connected to the Legacy receiver interface, configured to expect the standard Ethernet frames produced by the Legacy Frame Stimulus block
- A PLB Module that connects to the PLB interface of the core and contains simple state machines to perform initialization of configuration and interrupt management state machines.

Ethernet Frame Stimulus

The following files describe the Ethernet Frame Stimulus logic:

VHDL

<project_dir>/*<component_name>*/example_design/*tx_frame_stimulus.vhd*

Verilog

<project_dir>/*<component_name>*/example_design/*tx_frame_stimulus.v*

This module contains the logic to produce an Ethernet test frame. The MAC header fields of this frame are defined by generics (Destination Address, Source Address, Length/Type); the VLAN field is optional. Additionally, the length of the Ethernet frame can also be set using a generic.

The data field of the frame is designed to create a simple 8-bit binary counter that continues seamlessly across consecutive Ethernet frames. The Ethernet Frame Stimulus block is designed to produce frames at full line rate to fully stress the core.

Ethernet Frame Checker

The following files describe the Ethernet Frame Checker logic.

VHDL

<project_dir>/*<component_name>*/example_design/*rx_frame_checker.vhd*

Verilog

<project_dir>/*<component_name>*/example_design/*rx_frame_checker.v*

This module contains the logic to check a received Ethernet frame against expected parameters. The MAC header fields of this expected frame are defined by generics (Destination Address, Source Address, Length/Type); the VLAN field is optional. Additionally, the expected length of the Ethernet frame can also be set using a parameter.

The data field of the frame is expected to consist of a simple 8-bit binary counter which continues seamlessly across consecutive Ethernet frames.

This logic is designed to check against the frames generated by the tx_frame_stimulus module; identical parameters must be passed into both modules to obtain a match.

Loopback Module

The following files describe the Loopback module.

VHDL

<project_dir>/*<component_name>*/example_design/*temac_loopback_shim.vhd*

Verilog

<project_dir>/*<component_name>*/example_design/*temac_loopback_shim.v*

This logic implements a simple logic shim to provide a frame loopback function at the MAC client Interface. This logic does NOT implement a MAC and should be replaced with a real MAC in any real implementations.

PLB Module

The following files describe the logic for the PLB module.

VHDL

<project_dir>/*<component_name>*/example_design/*plb_client_logic.vhd*

Verilog

<project_dir>/*<component_name>*/example_design/*plb_client_logic.v*

The PLB module connects to the PLB interface of the core and performs the following functions:

- **Initialization**. A state machine writes to the RTC configuration space to set the RTC running at the correct frequency following reset/power-up.
- **PTP Timer Interrupt Service Routine**. When the `interrupt_ptp_timer` is asserted, a state machine requests transmission of a PTP sync frame, then clears the interrupt.
- **PTP Transmit Interrupt Service Routine**. When `interrupt_ptp_tx` is asserted (a PTP frame has been transmitted), the state machine reads from the PTP Tx Control/Status register to determine the type of PTP frame sent. If it was a sync frame, it then requests a follow-up frame to be sent. For any other PTP frame type, no action is taken. Reading from the PTP Tx Control/Status register clears the interrupt.
- **PTP Receive Interrupt Service Routine**. When `interrupt_ptp_rx` is asserted (a PTP frame has been received), the state machine reads from the PTP Rx Control/Status register to determine which of the PTP frame buffers the received frame will be stored in; this read also clears the interrupt. In this simple demonstration, nothing further is performed.

This functionality is related to the normal operation of a PTP clock master in that the logic results in a transmission of PTP Sync/Follow-Up pair of frames being sent periodically. However, the functionality is greatly simplified and none of the relevant variable PTP Sync/Follow-up fields are correctly set.

Note: The real intent for the PLB interface is for connection into the EDK environment; software drivers are provided to be run on an embedded processor, which performs full 802.1AS (Precise Timing Protocol (PTP)) functionality. See Chapter 13, “Software Drivers” for detailed information about the provided software drivers.

Demonstration Test Bench

Figure 15-2 illustrates the Ethernet AVB Endpoint demonstration test bench, a simple VHDL or Verilog program for exercising the example design and the core.

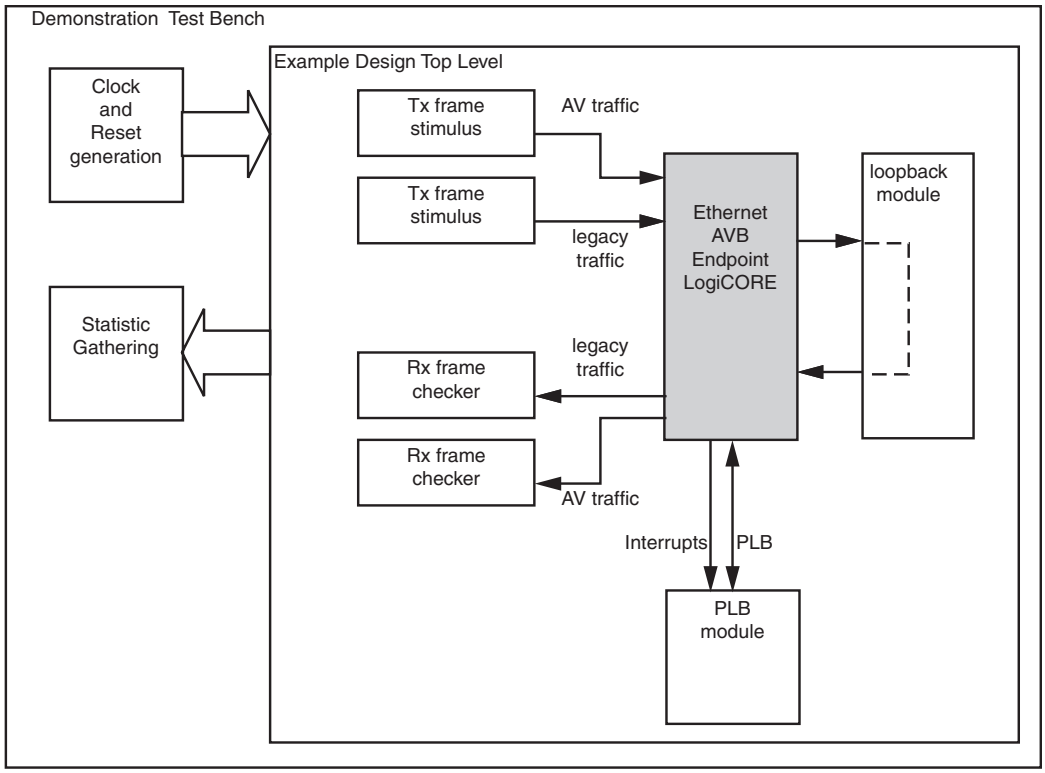

Figure 15-2: **Ethernet AVB Endpoint Demonstration Test Bench**

The following files describe the top level of the demonstration test bench:

VHDL

```
<project_dir>/<component_name>/simulation/demo_tb.vhd
```

Verilog

```
<project_dir>/<component_name>/simulation/demo_tb.v
```

The top-level test bench entity instantiates the example design for the core, which is the Device Under Test (DUT). The test bench provides clocks and resets, and gathers statistics for the duration of the simulation. A final statistic report is created at the end of the simulation run time that contains the following:

- The number of PTP frames transmitted and received
- The number of AV frames transmitted and received
- The number of legacy frames transmitted and received.
  All transmitted frame statistics should exactly match the received frame statistics for each particular frame type; if this is not the case, an error message is issued.
- Finally, the test bench estimates the percentage of overall Ethernet line rate consumed by each of the three types. This should illustrate the bandwidth policing functionality of the core, which should only allow the AV frames to consume a maximum of 75% of the overall bandwidth.

Customizing the Test Bench

Simulation Run Time

The default simulation run time is set to only 40 microseconds, which can be easily extended by editing the `simulation_run_time` constant, set near the top of the demonstration test bench file. For example, from the VHDL file:

```
------------------------------------------------------------------
-- **** The following value determines the simulations run time ****
------------------------------------------------------------------
constant simulation_run_time : time := 40000 ns;
```

The test bench allows the DUT to run until the simulation time is exceeded; after this, Ethernet frames already in the system are allowed to complete cleanly; then the test bench reports the final statistics and end.

Changing Frame Data

The Ethernet Frame Stimulus and Ethernet Frame Checker modules can be set to produce and check different Ethernet frames by changing the parameters sent to them. These parameters are set in the Top-Level Example Design HDL. Editing this file allows a Functional Simulation to immediately use the new settings. However, because these modifications require logical changes, the Implementation Scripts must be re-run on the design before running a Timing Simulation.

Please see the Top-Level Example Design HDL file for information about these frame-type parameters. As an example, the following syntax is taken from the Verilog version of the file and contains the syntax required to configure both the Legacy Ethernet Frame Stimulus and Ethernet Frame Checker modules:

```
//---------------------------------
// Configure the Legacy frames used in this example design (the
// following parameters can be edited)
//---------------------------------

// Use minimum sized Ethernet frames (64-bytes total length)
parameter [10:0]  LEGACY_FRAME_LENGTH  = 11'd64;

// Set the Destination Address to be AA-BB-CC-DD-EE-FF
parameter [47:0]  LEGACY_DEST_ADDR     = 48'hFFEEDDCCBBAA;

// Set the Destination Address to be 00-11-22-33-44-55
parameter [47:0]  LEGACY_SRC_ADDR      = 48'h554433221100;

// Do not use VLAN fields
parameter         LEGACY_HAS_VLAN      = 1'b0;

// VLAN fields are not used so the following parameter is n/a
parameter [15:0]  LEGACY_VLAN_DATA     = 16'h0000;

// Use a Generic Type field
parameter [15:0]  LEGACY_TYPE_FIELD    = 16'h8000;
```

Viewing the Simulation Wave Form

The Simulation Scripts for the selected simulator automatically selects signals of interest from within the DUT and adds them to the simulator wave window. These are organized into grouped interfaces, which are identified using section headings in the wave window.

Figure 15-3 illustrates the grouped interfaces selected for the Functional Simulation. The circled numbers represent the order in which they are displayed (the wave window section headings are also numbered to match Figure 15-3). Further signals of interest may be added as desired.

The signals added to the Timing Simulation are a subset of the ones used in the Functional Simulation. To summarize, the PLB interface is not viewed, due to synthesis/implementation optimization that occurs on these signals, the result of which merges signals and changes names.

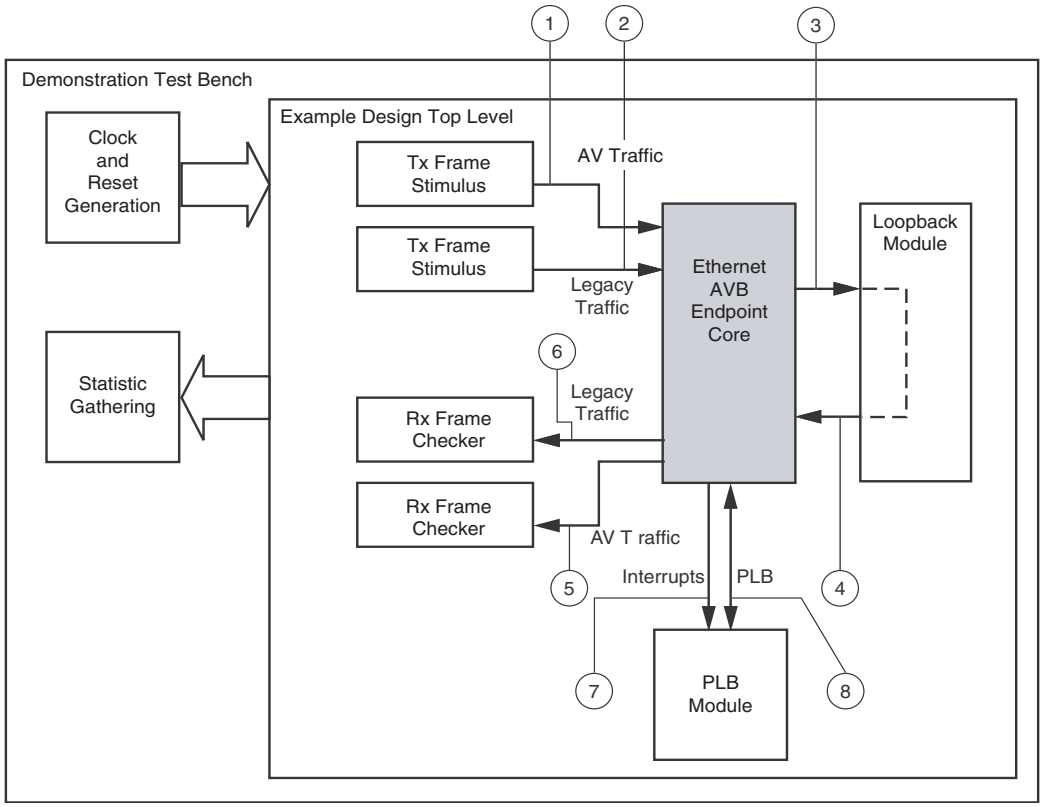

Figure 15-3: **Simulator Wave Window Contents**

XILINX®

Chapter 16

Detailed Example Design (EDK format)

This chapter provides detailed information about the core when generated in the Standard Embedded Development Kit (EDK) format, including a description of files and the directory structure generated. This option is selected from page 1 of the customization GUI.

Please refer instead to Chapter 15, "Detailed Example Design (Standard Format)." when requiring the Standard CORE Generator™ software format.

- **<project directory>**
  Top-level project directory; name is user-defined.
  - <project directory>/<component name>
    Core release notes file
    - <component name>/doc
      Product documentation
    - MyProcessorIPLib/pcores/eth_avb_endpoint_v2_04_a
      core netlist and HDL for the pcore
      - pcores/eth_avb_endpoint_v2_04_a/data
        Data files for automatic integration into Xilinx Platform Studio
      - pcores/eth_avb_endpoint_v2_04_a/hdl/vhdl
        VHDL wrapper file for the core netlist to enable integration into Platform Studio
      - pcores/eth_avb_endpoint_v2_04_a/netlist
        The Ethernet AVB Endpoint core netlist
    - MyProcessorIPLib/drivers/avb_v2_04_a
      Software Device Drivers for the pcore
      - drivers/avb_v2_04_a/data
        Data files for automatic integration into Xilinx Platform Studio
      - drivers/avb_v2_04_a/examples
        An application example using the low-level driver files
      - drivers/avb_v2_04_a/src
        Low-level driver source C files

Directory and File Contents

The core directories and their associated files are defined in the following tables.

<project directory>

The project directory contains all the CORE Generator software project files.

Table 16-1: **Project Directory**

Name	Description
<project_dir>	
<component_name>.ngc	Top-level netlist. This is instantiated by the Verilog or VHDL example design.
<component_name>.xco	Log file that records the settings used to generate a core. An XCO file is generated by the CORE Generator software for each core that it creates in the current project directory. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	List of files delivered with the core.

Back to Top

<project directory>/<component name>

The <component name> directory contains the release notes file provided with the core, which may include last-minute changes and updates.

Table 16-2: **Component Name Directory**

Name	Description
<project_dir>/<component_name>	
eth_avb_endpoint_readme.txt	Core release notes file.

Back to Top

<component name>/doc

The doc directory contains the PDF documentation provided with the core.

Table 16-3: **Doc Directory**

Name	Description
<project_dir>/<component_name>/doc	
eth_avb_endpoint_ds677.pdf	Ethernet AVB Endpoint Data Sheet
eth_avb_endpoint_ug492.pdf	Ethernet AVB Endpoint User Guide

Back to Top

<component name>/MyProcessorIPLib

This is the route directory which should be imported into the Xilinx Embedded Development Kit.

MyProcessorIPLib/pcores/eth_avb_endpoint_v2_04_a

A directory containing the pcore HDL and netlist hardware components for the Ethernet AVB Endpoint core and associated supporting files.

pcores/eth_avb_endpoint_v2_04_a/data

The driver data directory contains the data files for automatic integration into Platform Studio.

Table 16-4: **Driver Data Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/pcores/ eth_avb_endpoint_v2_04/data	
eth_avb_endpoint_v2_1_0.bbd	Black Box description file for the core netlist.
eth_avb_endpoint_v2_1_0.mpd	Microprocessor Description file for the pcore (contains a list and definition of ports)
eth_avb_endpoint_v2_1_0.pao	Peripheral Analyze Order file (containing a list of HDL sources requiring synthesis.

Back to Top

pcores/eth_avb_endpoint_v2_04_a/hdl/vhdl

Contains a VHDL wrapper file for the core netlist to enable integration into Platform Studio.

Table 16-5: **Driver Data Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/pcores/ eth_avb_endpoint_v2_04/hdl/vhdl	
eth_avb_endpoint.vhd	This is a wrapper file around the Ethernet AVB Endpoint netlist to enable dynamic PLB address assignment from within Platform Studio.

Back to Top

pcores/eth_avb_endpoint_v2_04_a/netlist

The pcore netlist directory contains the netlist for the core that was synthesized during core generation.

Table 16-6: **pcore netlist Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/pcores/ eth_avb_endpoint_v2_04/netlist	
<component_name>.ngc	Netlist for the core that was synthesized during core generation

Back to Top

MyProcessorIPLib/drivers/avb_v2_04_a

A directory containing the software device drivers for the Ethernet AVB Endpoint core and associated supporting files.

drivers/avb_v2_04_a/data

The driver data directory contains the data files for automatic generation of parameter specific files when integrated into Platform Studio.

Table 16-7: **Driver Data Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/drivers/avb_v2_04/data	
avb_v2_1_0.mdd	Current MDD file used, including the version of the tools interface.
avb_v2_1_0.tcl	Used to provide design rule checks within Xilinx Platform Studio.

Back to Top

drivers/avb_v2_04_a/examples

The driver examples directory contains an application example using the low-level driver files.

Table 16-8: **Driver Example Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/drivers/avb_v2_04/examples	
xavb_example.c	Contains a very basic example design of using the AVB driver.

Back to Top

drivers/avb_v2_04_a/src

The driver source (src) directory contains the low-level driver source C files.

Table 16-9: **Driver Source Directory**

Name	Description
<project_dir>/<component_name>/MyProcessorIPLib/drivers/avb_v2_04/src	
Makefile	Makefile to compile the drivers; used by Platform Studio.
xavb.h	Main header file for the XAvb driver. The file provides the constants, type definitions and function templates which are required to initialize and run the IEEE802.1AS Precise Timing Protocol (PTP). This defines the level 1 device driver for the Ethernet AVB Endpoint core.
xavb_g.c	Contains a configuration structure that holds all the configuration values required, per single instance, of the device driver.
xavb.c	Provides the top-level function calls for the Ethernet AVB Endpoint level 1 device driver.
xavb_ptp_packets.c	Provides the functions which are required for the creation of PTP frames for transmission and for the decode of received PTP frames.
xavb_ptp_bmca.c	Provides the functions which are required for the PTP Best Master Clock Algorithm (BMCA).
xavb_rtc_sync.c	Provides the functions which are required to synchronize the local version of the Real Time Counter (RTC), when operating as a slave, to that of the network clock master.
xavb_hw.h	Contains all the constant definitions and the bare minimum of functions / function templates which are required for register read/write access. This defines the low-level 0 device driver for the Ethernet AVB Endpoint core.
xavb_hw.c	This file partners the xavb_hw.h header file and implements the functions for which avb_hw.h contained a template.

Back to Top

Importing the Ethernet AVB Endpoint Core into the Embedded Development Kit (EDK)

You can import a generated Ethernet AVB Endpoint netlist into an EDK project by following the usual steps to import a black box IP. See the Xilinx Platform Studio documentation for information.

After importing the generated netlist, the drivers can also be linked into the software application. See "Software System Integration" in Chapter 13 for more information.

XILINX®

Appendix A

RTC Time Stamp Accuracy

Time Stamp Accuracy

The accuracy of the time stamps, taken by sampling the "Real Time Clock" (RTC) whenever PTP frames are transmitted or received, is essential to the Precise Timing Protocol across the network link. For this reason, the time stamps are performed in hardware. Despite this, time stamp inaccuracies can be introduced from two sources:

- "RTC Real Time Instantaneous Error"
- "RTC Sampling Error"

Following this discussion, we then consider the "Accuracy Resulting from the Combined Errors."

RTC Real Time Instantaneous Error

Figure A-1 illustrates a RTC implementation which uses a 40 ns clock period as its clock source (this is worst case). Therefore, the controlled frequency RTC will only be updated every 40ns. Because the concept of a RTC is a continuous measurement of time, the implementation of the RTC illustrated in Figure A-1 is only accurate immediately after an update. During the 40 ns update cycle, the error accrues linearly to a maximum of 40 ns. This behavior is periodic as illustrated.

In Figure A-1, two time stamps of the RTC are sampled. The figure shows that the accuracy is variable. For example:

- The 1st time stamp is requested at 119 ns. However, the RTC has yet to update and so the sample taken will be of 80 ns. This has an inaccuracy of 39 ns.
- The 2nd time stamp is requested at 201 ns. The RTC has recently updated and so the sample taken will be of 200. This has an inaccuracy of 1 ns.

The maximum RTC inaccuracy, per time stamp sample, is equal to the period of the RTC reference clock (in this example 40 ns). By using a high frequency RTC reference clock, a high degree of accuracy can be obtained.

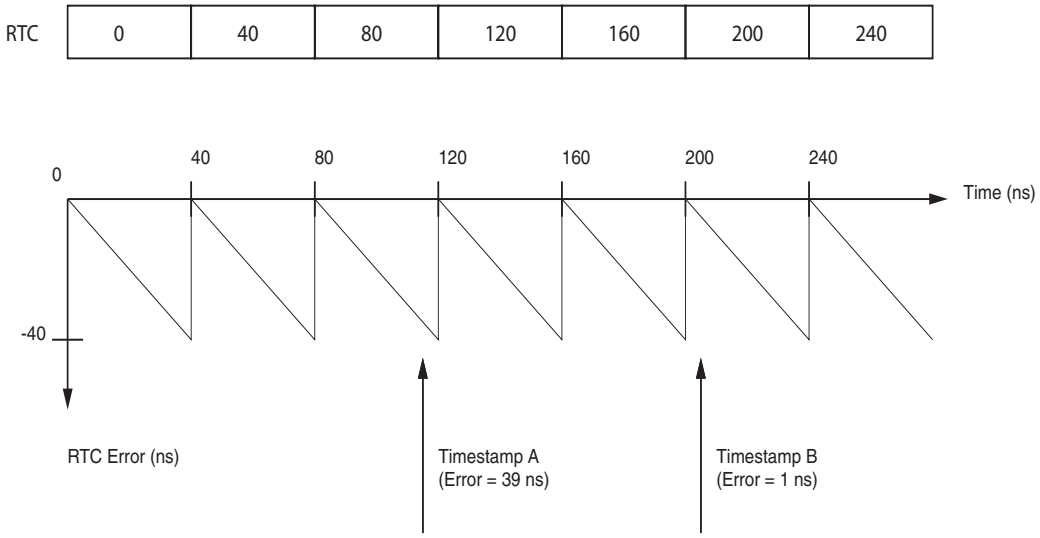

Figure A-1: **RTC Periodic Error**

RTC Sampling Error

It has to be assumed that the RTC reference clock is of a different frequency to the MAC transmitted and receiver clocks. Therefore, the RTC sampling logic has to be asynchronous.

There are a number of methods to obtain a time stamp across an asynchronous clock boundary. The simplest method, is to simply pass a toggle signal from the Tx/Rx domain into the RTC reference clock domain whenever a time stamp is required. This method should only result in an uncertainty of one cycle: the logic is illustrated in Figure A-2.

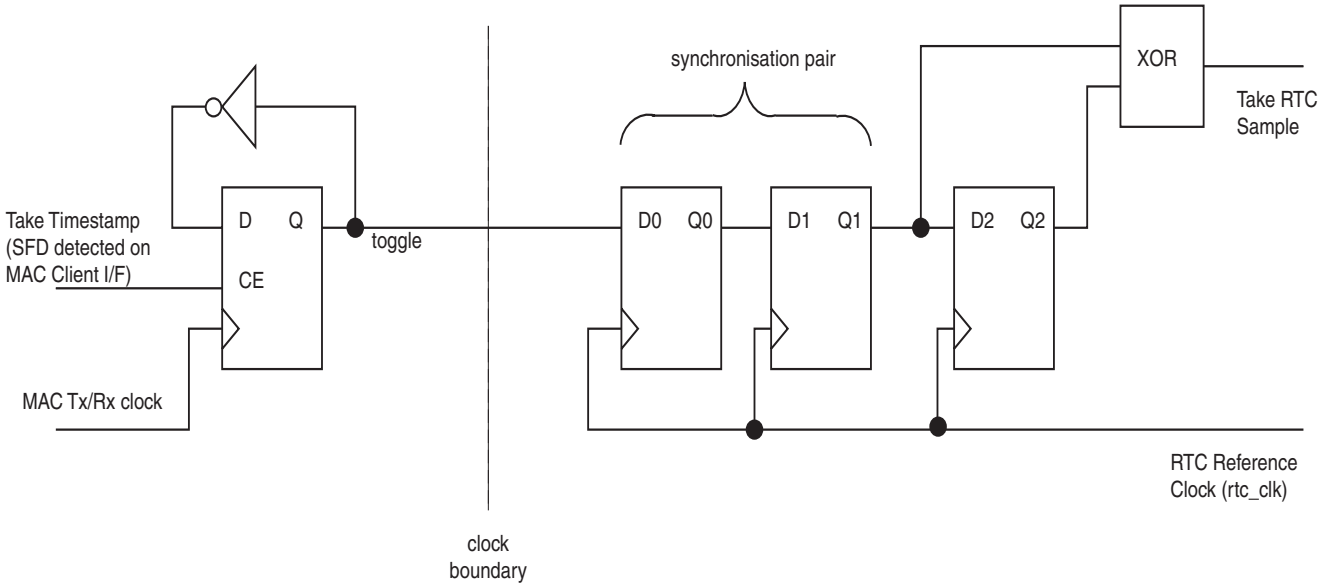

Figure A-2: **RTC Sampling Logic**

In Figure A-2, the `Sample Timestamp` signal is generated whenever the Tx/Rx time stamp position is detected (see "Time Stamp Sampling Position of MAC Frames"). From this, a `toggle` signal is generated as illustrated, and this is passed across the clock domain from Tx/Rx MAC clock to the RTC reference clock domain.

When in the RTC clock domain, the `toggle` signal is re-clocked using the two synchronization flip-flops illustrated. After this, an edge detection circuit is used to determine that the RTC should be sampled.

The single clock period of uncertainty arises from the behavior of the first synchronization flip-flop. Figure A-3 illustrates that the rising edge of the `toggle` signal can occur very close to the clock edge of the RTC reference clock. It is possible that the setup timing of this flip-flop could be violated, resulting in uncertainty as to whether a logic '0' or a logic '1' will be sampled. If the flip-flop samples logic '1', the result is Timing Case 1; if the flip-flop samples logic '0', Timing Case 2 results.

The overall result of this is to obtain a single Reference Clock Period of uncertainty in the captured time stamp value.

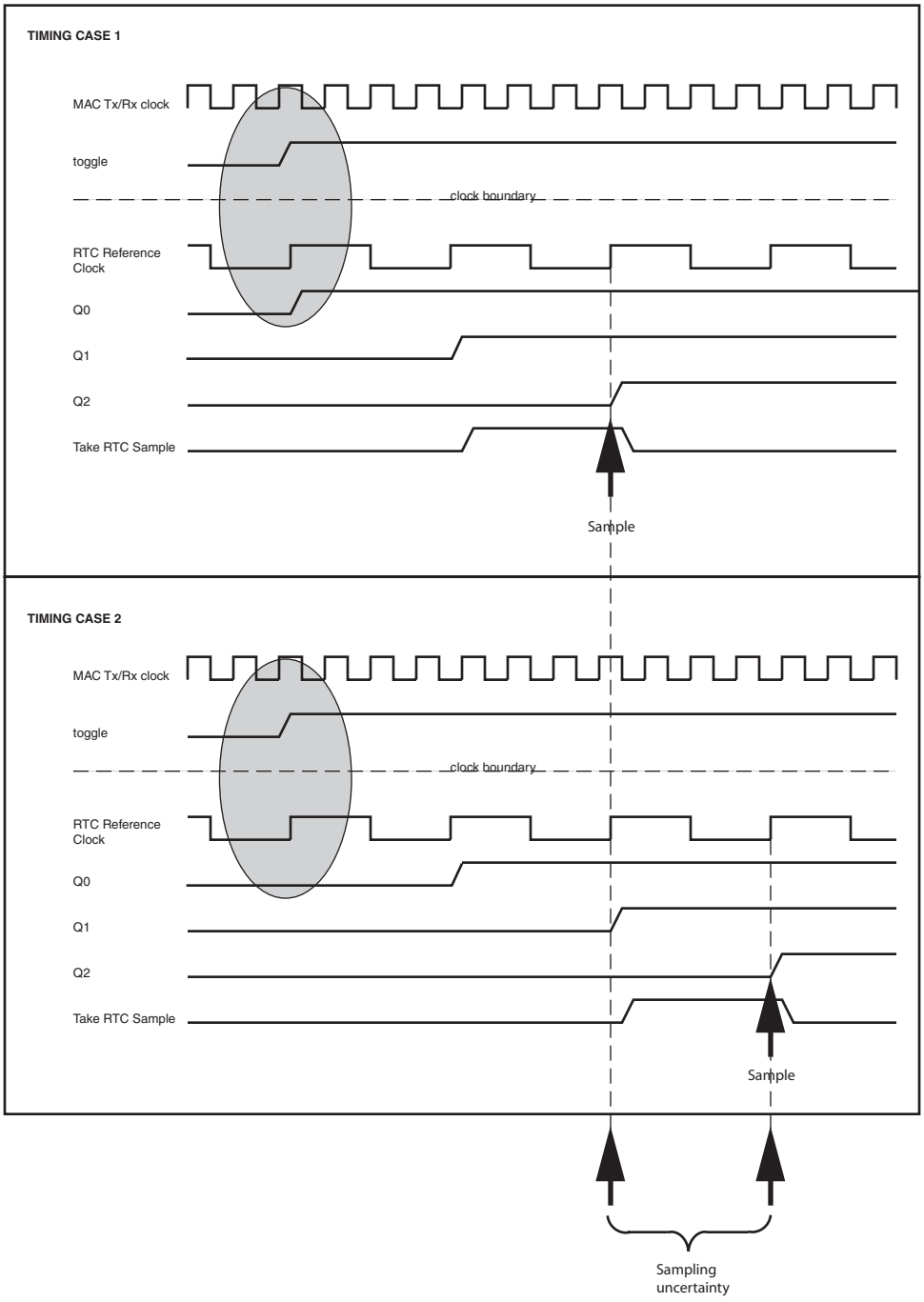

Figure A-3: **Sampling Position Uncertainty**

Accuracy Resulting from the Combined Errors

The section "RTC Real Time Instantaneous Error" describes how a maximum error of one RTC reference clock period can result as a consequence of the RTC itself. The section "RTC Sampling Error" describes how the position of the time stamp request, as observed in the RTC reference clock domain, can result in one RTC reference clock period of uncertainty. Figure A-4 attempts to illustrate the result of the combination of these two types of error. Again, the worst case clock period of 40 ns is illustrated.

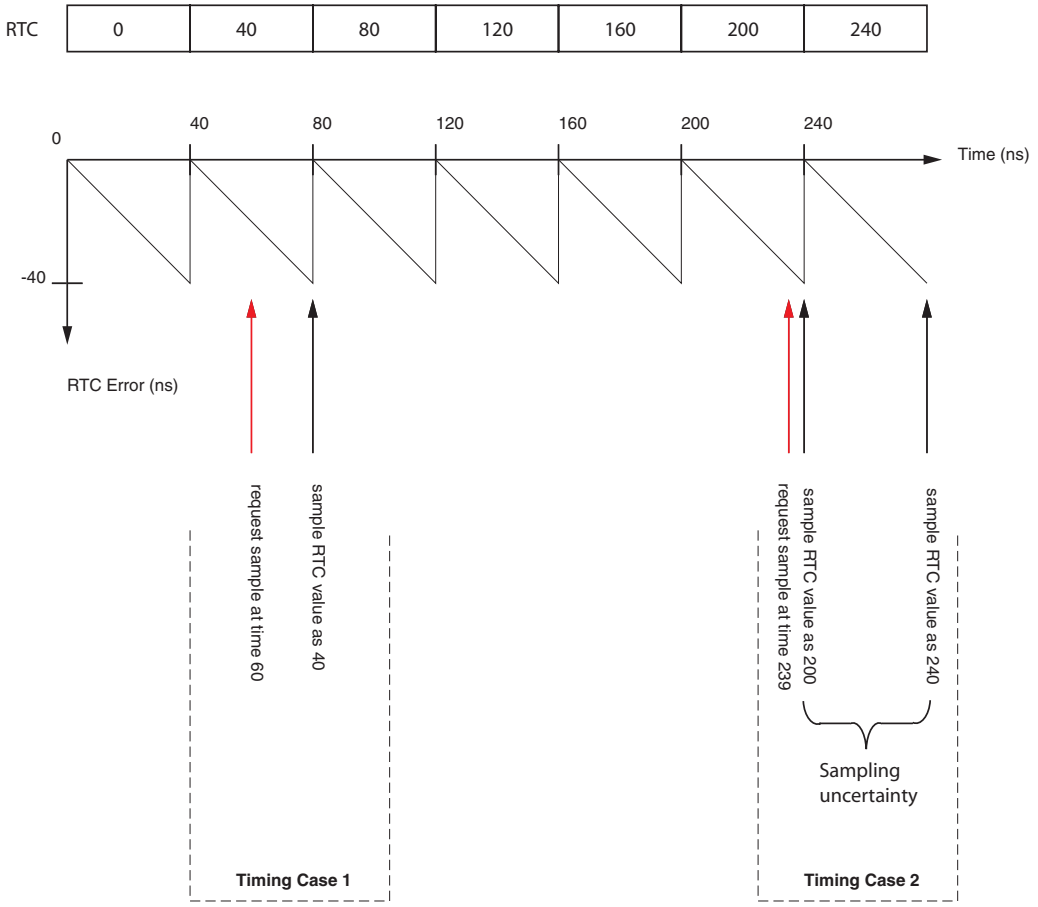

Figure A-4: **Overall Time Stamp Accuracy**

In Figure A-4, two time stamps of the RTC are sampled. The figure shows that the accuracy is variable. For example:

- The request for the 1st time stamp is made at 60 ns. Because the time to the next RTC reference clock is 20 ns, this will not violate the setup time for the 1st synchronization flip-flop in Figure A-2. Therefore, on the next RTC reference clock, the sample will be taken as 40 ns (resulting in an error of 20 ns which is entirely due to the "RTC Real Time Instantaneous Error").
- The request for the 2nd time stamp is made at 239 ns. This is very close to the rising edge of the 1st synchronization flip-flop in Figure A-2, so the situation is unpredictable:

- If the flip-flop samples the new value, then Timing Case 1 results. The RTC is sampled as 200 (resulting in an error of 39 ns which is entirely due to the "RTC Real Time Instantaneous Error").
- If the flip-flop samples the old value, then Timing Case 2 results. The RTC is sampled 1 RTC reference clock period later as 240 (resulting in an error of only 1 ns).

Hopefully these examples have illustrated that the timing uncertainty in the asynchronous sampling circuit has not resulted in any additional error.

The maximum inaccuracy, per time stamp sample, is still equal to the period of the RTC reference clock (in this example 40 ns). By using a high frequency RTC reference clock, a high degree of accuracy can be obtained. For example, when using a 125 MHz clock source for the RTC, the maximum time stamp error will be 8 ns or less.